

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
please do not report the images to the
Image Problem Mailbox.**

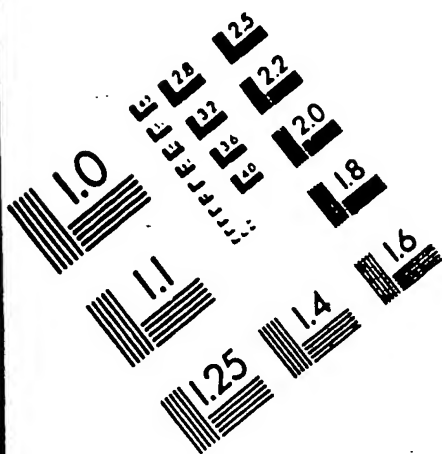
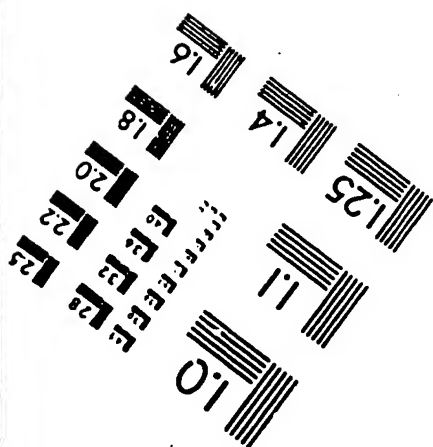
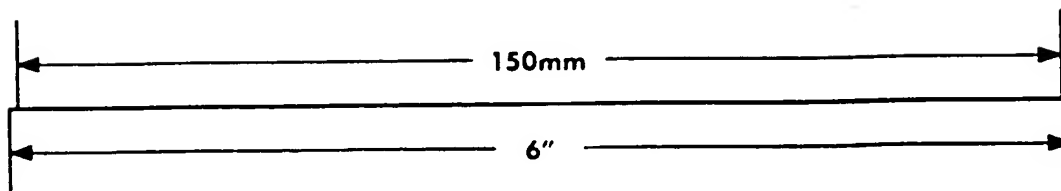
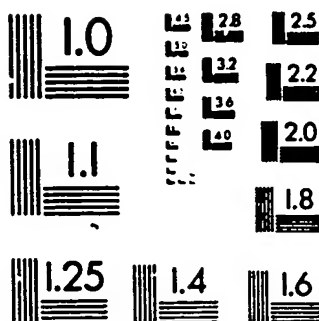
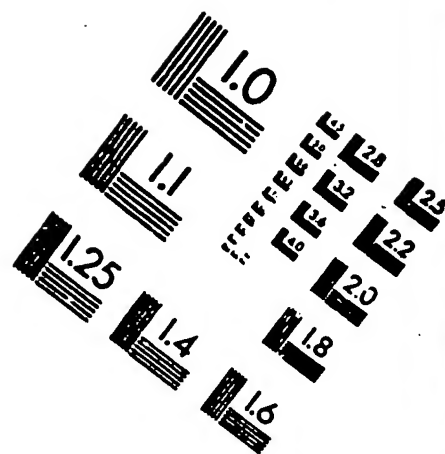
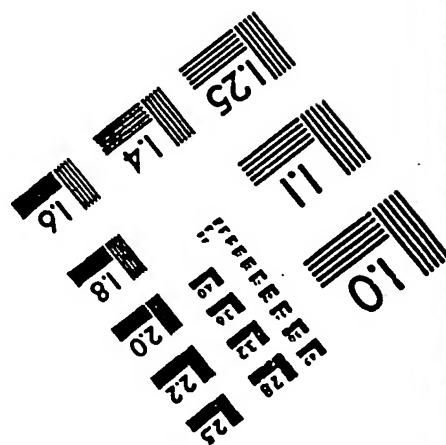


IMAGE EVALUATION TEST TARGET (MT-3)



PHOTOGRAPHIC SCIENCES CORPORATION
770 BASKET ROAD
P.O. BOX 338
WEBSTER, NEW YORK 14580
(716) 265-1600



microunity

Zeus System Architecture

COPYRIGHT 1998 MICROUNITY SYSTEMS ENGINEERING, INC. ALL RIGHTS RESERVED.



MicroUnity

**Craig Hansen
Chief Architect**

MicroUnity Systems Engineering, Inc.
475 Potrero Avenue
Sunnyvale, CA 94086.4118
Phone: 408.734.8100
Fax: 408.734.8136
email: craig@microunity.com
<http://www.microunity.com>

PBGNT#	IO	Private Bus GRANT is driven between Primary and Dual processors to indicate that bus arbitration has completed, granting a new master access to the bus.
PBREQ#	IO	Private Bus REQuest is driven between Primary and Dual processors to request a new master access to the bus.
PCD	O	Page Cache Disable is driven with address to indicate a not cacheable transaction.
PCHK#	O	Parity CHeck is asserted (driven low) two bus clocks after data appears with odd parity on enabled bytes.
PHIT#	IO	Private HIT is driven between Primary and Dual processors to indicate that the current read or write transaction addresses a valid cache sub-block in the slave processor.
PHITM#	IO	Private HIT Modified is driven between Primary and Dual processors to indicate that the current read or write transaction addresses a modified cache sub-block in the slave processor.
PICCLK	I	Programmable Interrupt Controller CLoCK is not implemented.
PICD1..PICD0	IO	Programmable Interrupt Controller Data is not implemented.
PEN#	I	Parity Enable, if active on the data cycle, allows a parity error to cause a bus error machine check.
PM1..PM0	O	Performance Monitoring is an emulator signal.
PRDY	O	Probe ReADY is not implemented.
PWT	O	Page Write Through is driven with address to indicate a not write allocate transaction.
R/S#	I	Run/Stop is not implemented.
RESET	I	RESET causes a processor reset.
SCYC	O	Split CYCLE is asserted during bus lock to indicate that more than two transactions are in the series of bus transactions.
SMI#	I	System Management Interrupt is an emulator signal.
SMIACK#	O	System Management Interrupt ACTive is an emulator signal.
STPCLK#	I	SToP CLoCK is an emulator signal.
TCK	I	Test CLoCK follows IEEE 1149.1.
TDI	I	Test Data Input follows IEEE 1149.1.
TDO	O	Test Data Output follows IEEE 1149.1.
TMS	I	Test Mode Select follows IEEE 1149.1.
TRST#	I	Test ReSeT follows IEEE 1149.1.
VCC2	I	VCC of 2.8V at 25 pins
VCC3	I	VCC of 3.3V at 28 pins
VCC2DET#	O	VCC2 DETect sets appropriate VCC2 voltage level.
VSS	I	VSS supplied at 53 pins
W/R#	O	Write/Read is driven with address to indicate write

		vs. read transaction.
WB/WT#	1	Write Back/Write Through is returned to indicate that data is permitted to be cached as write back.

Electrical Specifications

These preliminary electrical specifications provide AC and DC parameters that are required for "Super Socket 7" compatibility.

Clock rate	66 MHz		75 MHz		100 MHz		133 MHz		
Parameter	min	max	min	max	min	max	min	max	unit
CLK frequency	33.3	66.7	37.5	75	50	100		133	MHz
CLK period	15.0	30.0	13.3	26.3	10.0	20.0			ns
CLK high time (2ZV)	4.0		4.0		3.0				ns
CLK low time (50.8V)	4.0		4.0		3.0				ns
CLK rise time (0.8V->2V)	0.15	1.5	0.15	1.5	0.15	1.5			ns
CLK fall time (2V->0.8V)	0.15	1.5	0.15	1.5	0.15	1.5			ns
CLK period stability		250		250		250			ps

A31..3 valid delay	1.1	6.3	1.1	4.5	1.1	4.0		ns
A31..3 float delay		10.0		7.0		7.0		ns
ADS# valid delay	1.0	6.0	1.0	4.5	1.0	4.0		ns
ADS# float delay		10.0		7.0		7.0		ns
ADSC# valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
ADSC# float delay		10.0		7.0		7.0		ns
AP valid delay	1.0	8.5	1.0	5.5	1.0	5.5		ns
AP float delay		10.0		7.0		7.0		ns
APCHK# valid delay	1.0	8.3	1.0	4.5	1.0	4.5		ns
BE7..0# valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
BE7..0# float delay		10.0		7.0		7.0		ns
BP3..0 valid delay	1.0	10.0						ns
BREQ valid delay	1.0	8.0	1.0	4.5	1.0	4.0		ns
CACHE# valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
CACHE# float delay		10.0		7.0		7.0		ns
D/C# valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
D/C# float delay		10.0		7.0		7.0		ns
D63..0 write data valid delay	1.3	7.5	1.3	4.5	1.3	4.5		ns
D63..0 write data float delay		10.0		7.0		7.0		ns
DP7..0 write data valid delay	1.3	7.5	1.3	4.5	1.3	4.5		ns
DP7..0 write data float delay		10.0		7.0		7.0		ns
FERR# valid delay	1.0	8.3	1.0	4.5	1.0	4.5		ns
HIT# valid delay	1.0	6.8	1.0	4.5	1.0	4.0		ns
HITM# valid delay	1.1	6.0	1.1	4.5	1.1	4.0		ns
HLDA valid delay	1.0	6.8	1.0	4.5	1.0	4.0		ns
IERR# valid delay	1.0	8.3						ns
LOCK# valid delay	1.1	7.0	1.1	4.5	1.1	4.0		ns
LOCK# float delay		10.0		7.0		7.0		ns
M/IO# valid delay	1.0	5.9	1.0	4.5	1.0	4.0		ns
M/IO# float delay		10.0		7.0		7.0		ns
PCD valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
PCD float delay		10.0		7.0		7.0		ns
PCHK# valid delay	1.0	7.0	1.0	4.5	1.0	4.5		ns
PM1..0 valid delay	1.0	10.0						ns
PRDY valid delay	1.0	8.0						ns
PWT valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
PWT float delay		10.0		7.0		7.0		ns
SCYC valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
SCYC float delay		10.0		7.0		7.0		ns
SMACT# valid delay	1.0	7.3	1.0	4.5	1.0	4.0		ns
W/R# valid delay	1.0	7.0	1.0	4.5	1.0	4.0		ns
W/R# float delay		10.0		7.0		7.0		ns

Zeus System Architecture

Tue, Aug 17, 1999

Bus interface
Electrical Specifications

A31.5 setup time	6.0	3.0	3.0	ns
A31.5 hold time	1.0	1.0	1.0	ns
A20M# setup time	5.0	3.0	3.0	ns
A20M# hold time	1.0	1.0	1.0	ns
AHOLD setup time	5.5	3.5	3.5	ns
AHOLD hold time	1.0	1.0	1.0	ns
AP setup time	1.7	1.7	1.7	ns
AP hold time	1.0	1.0	1.0	ns
BOFF# setup time	5.5	3.5	3.5	ns
BOFF# hold time	1.0	1.0	1.0	ns
BRDY# setup time	5.0	3.0	3.0	ns
BRDY# hold time	1.0	1.0	1.0	ns
BRDYC# setup time	5.0	3.0	3.0	ns
BRDYC# hold time	1.0	1.0	1.0	ns
BUSCHKE setup time	5.0	3.0	3.0	ns
BUSCHKE hold time	1.0	1.0	1.0	ns
D63.0 read data setup time	2.8	1.7	1.7	ns
D63.0 read data hold time	1.5	1.5	1.5	ns
DP7.0 read data setup time	2.8	1.7	1.7	ns
DP7.0 read data hold time	1.5	1.5	1.5	ns
EADS# setup time	5.0	3.0	3.0	ns
EADS# hold time	1.0	1.0	1.0	ns
EWBE# setup time	5.0	1.7	1.7	ns
EWBE# hold time	1.0	1.0	1.0	ns
FLUSH# setup time	5.0	1.7	1.7	ns
FLUSH# hold time	1.0	1.0	1.0	ns
FLUSH# async pulse width	2	2	2	CLK
HOLD setup time	5.0	1.7	1.7	ns
HOLD hold time	1.5	1.5	1.5	ns
IGNNE# setup time	5.0	1.7	1.7	ns
IGNNE# hold time	1.0	1.0	1.0	ns
IGNNE# async pulse width	2	2	2	CLK
INIT setup time	5.0	1.7	1.7	ns
INIT hold time	1.0	1.0	1.0	ns
INIT async pulse width	2	2	2	CLK
INTR setup time	5.0	1.7	1.7	ns
INTR hold time	1.0	1.0	1.0	ns
INV setup time	5.0	1.7	1.7	ns
INV hold time	1.0	1.0	1.0	ns
KEN# setup time	5.0	3.0	3.0	ns
KEN# hold time	1.0	1.0	1.0	ns
NA# setup time	4.5	1.7	1.7	ns
NA# hold time	1.0	1.0	1.0	ns
NMI setup time	5.0	1.7	1.7	ns
NMI hold time	1.0	1.0	1.0	ns
NMI async pulse width	2	2	2	CLK

PEN# setup time	4.8		1.7		1.7			ns
PEN# hold time	1.0		1.0		1.0			ns
R/S# setup time	5.0		1.7		1.7			ns
R/S# hold time	1.0		1.0		1.0			ns
R/S# async pulse width	2		2		2			CLK
SM# setup time	5.0		1.7		1.7			ns
SM# hold time	1.0		1.0		1.0			ns
SM# async pulse width	2		2		2			CLK
STPCLK# setup time	5.0		1.7		1.7			ns
STPCLK# hold time	1.0		1.0		1.0			ns
WB/W# setup time	4.5		1.7		1.7			ns
WB/W# hold time	1.0		1.0		1.0			ns

RESET setup time	5.0		1.7		1.7			ns
RESET hold time	1.0		1.0		1.0			ns
RESET pulse width	15		15		15			CLK
RESET active	1.0		1.0		1.0			ms
BF2..0 setup time	1.0		1.0		1.0			ms
BF2..0 hold time	2		2		2			CLK
BRDY# hold time	1.0		1.0		1.0			ns
BRDY# setup time	2		2		2			CLK
BRDY# hold time	2		2		2			CLK
FLUSH# setup time	5.0		1.7		1.7			ns
FLUSH# hold time	1.0		1.0		1.0			ns
FLUSH# setup time	2		2		2			CLK
FLUSH# hold time	2		2		2			CLK

Zeus System Architecture

Tue, Aug 17, 1999

Fus interface
Electrical Specifications

PBREQ# flight time	0	2.0							ns
PBGNT# flight time	0	2.0							ns
PHIT# flight time	0	2.0							ns
PHITM# flight time	0	1.8							ns
A31.5 setup time	3.7								ns
A31.5 hold time	0.8								ns
D/C# setup time	4.0								ns
D/C# hold time	0.8								ns
W/R# setup time	4.0								ns
W/R# hold time	0.8								ns
CACHE# setup time	4.0								ns
CACHE# hold time	1.0								ns
LOCK# setup time	4.0								ns
LOCK# hold time	0.8								ns
SCYC setup time	4.0								ns
SCYC hold time	0.8								ns
ADS# setup time	5.8								ns
ADS# hold time	0.8								ns
M/IO# setup time	5.8								ns
M/IO# hold time	0.8								ns
HIT# setup time	6.0								ns
HIT# hold time	1.0								ns
HITM# setup time	6.0								ns
HITM# hold time	0.7								ns
HLDA setup time	6.0								ns
HLDA hold time	0.8								ns
DPEN# valid time		10.0							CLK
DPEN# hold time	2.0								CLK
D/P# valid delay (primary)	1.0	8.0							ns

TCK frequency		25				25			MHz
TCK period	40.0				40.0				ns
TCK high time ($\geq 2V$)	14.0				14.0				ns
TCK low time ($\leq 0.8V$)	14.0				14.0				ns
TCK rise time ($0.8V \rightarrow 2V$)		5.0				5.0			ns
TCK fall time ($2V \rightarrow 0.8V$)		5.0				5.0			ns
TRST# pulse width	30.0				30.0				ns

TDI setup time	5.0				5.0				ns
TDI hold time	9.0				9.0				ns
TMS setup time	5.0				5.0				ns
TMS hold time	9.0				9.0				ns
TDO valid delay	3.0	13.0			3.0	13.0			ns
TDO float delay		16.0				16.0			ns
all outputs valid delay	3.0	13.0			3.0	13.0			ns
all outputs float delay		16.0				16.0			ns
all inputs setup time	5.0				5.0				ns
all inputs hold time	9.0				9.0				ns

Bus Control Register

The Bus Control Register provides direct control of Emulator signals, selecting output states and active input states for these signals.

The layout of the Bus Control Register is designed to match the assignment of signals to the Event Register.

number	control
0	Reserved
1	A20M# active level
2	BF0 active level
3	BF1 active level
4	BF2 active level
5	BUSCHK active level
6	FLUSH# active level
7	FRCMC# active level
8	IGNNE# active level
9	INIT active level
10	INTR active level
11	NMI active level
12	SMI# active level
13	STPCLK# active level
14	CPUTYP active at reset
15	DPEN# active at reset
16	FLUSH# active at reset
17	INIT active at reset
31..18	Reserved
32	Bus lock
33	Split cycle
34	BP0 output
35	BP1 output
36	BP2 output
37	BP3 output
38	FERR# output
39	IERR# output
40	PM0 output
41	PM1 output
42	SMIACK# output
63..43	Reserved

Emulator signals

Several of the signals, A20M#, INIT, NMI, SMI#, STPCLK#, IGNNE# are inputs that have purposes primarily defined by the needs of x86 processor emulation. They have no direct purpose in the Zeus processor, other than to signal an event, which is handled by software. Each of these signals is an input sampled on the rising edge of each bus clock, if the input signal matches the active level specified in the bus control register, the corresponding bit in the event register is set. The bit in the event register remains set even if the signal is no longer active, until cleared by software. If the event register bit is cleared by software, it is set again on each bus clock that the signal is sampled active.

A20M#

A20M# (address bit 20 mask inverted), when asserted (low), directs an x86 emulator to generate physical addresses for which bit 20 is zero.

The A20M# bit of the bus control register selects which level of the A20M# signal will generate an event in the A20M# bit of the event register. Clearing (to 0) the A20M# bit of the bus control register will cause the A20M# bit of the event register to be set when the A20M# signal is asserted (low).

Asserting the A20M# signal causes the emulator to modify all current TB mappings to produce a zero value for bit 20 of the byte address. The A20M# bit of the bus control register is then set (to 1) to cause the A20M# bit of the event register to be set when the A20M# signal is released (high).

Releasing the A20M# signal causes the emulator to restore the TB mapping to the original state. The A20M# bit of the bus control register is then cleared (to 0) again, to cause the A20M# bit of the event register to be set when the A20M# signal is asserted (low).

INIT

INIT (initialize) when asserted (high), directs an x86 emulator to begin execution of the external ROM BIOS.

The INIT bit of the bus control register is normally set (to 1) to cause the INIT bit of the event register to be set when the INIT signal is asserted (high).

INTR

INTR (maskable interrupt) when asserted (high), directs an x86 emulator to simulate a maskable interrupt by generating two hopped interrupt acknowledge special cycles. External hardware will normally release the INTR signal between the first and second interrupt acknowledge special cycle.

The INTR bit of the bus control register is normally set (to 1) to cause the INTR bit of the event register to be set when the INTR signal is asserted (high).

NMI

NMI (non-maskable interrupt) when asserted (high), directs an x86 emulator to simulate a non-maskable interrupt. External hardware will normally release the NMI signal.

The NMI bit of the bus control register is normally set (to 1) to cause the NMI bit of the event register to be set when the NMI signal is asserted (high).

SMI#

SMI# (system management interrupt inverted) when asserted (low), directs an x86 emulator to simulate a system management interrupt by flushing caches and saving registers, and asserting (low) **SMIACK#** (system management interrupt active inverted). External hardware will normally release the **SMI#**.

The **SMI#** bit of the bus control register is normally cleared (to 0) to cause the **SMI#** bit of the event register to be set when the **SMI#** signal is asserted (low).

STPCLK#

STPCLK# (stop clock inverted) when asserted (low), directs an x86 emulator to simulate a stop clock interrupt by flushing caches and saving registers, and performing a stop grant special cycle.

The **STPCLK#** bit of the bus control register is normally cleared (to 0) to cause the **STPCLK#** bit of the event register to be set when the **STPCLK#** signal is asserted (low).

Software must set (to 1) the **STPCLK#** bit of the bus control register to cause the **STPCLK#** bit of the event register to be set when the **STPCLK#** signal is released (high) to resume execution. Software must cease producing bus operations after the stop grant special cycle. Usually, software will use the **HLT** instruction in all threads to cease performing operations. The processor **PIU** continues to operate, and the processor must still sample **INIT**, **INTR**, **RESET**, **NMI**, **SMI#** (to place them in the event register) and respond to **RESET** and inquire and snout transactions, so long as the bus clock continues operating.

The bus clock itself cannot be stopped until the stop grant special cycle. If the bus clock is stopped, it must stop in the low (0) state. The bus clock must be operating at frequency for at least 1 ms before releasing **STPCLK#** or releasing **RESET**. While the bus clock is stopped, the processor does not sample inputs or responds to **RESET** or inquire or snout transactions.

External hardware will normally release **STPCLK#** when it is desired to resume execution. The processor should respond to the **STPCLK#** bit in the event register by scheduling one or more threads.

IGNNE#

IGNNE# (address bit 20 mask inverted), when asserted (low), directs an x86 emulator to ignore numeric errors.

The **IGNNE#** bit of the bus control register selects which level of the **IGNNE#** signal will generate an event in the **IGNNE#** bit of the event register. Clearing (to 0) the **IGNNE#** bit of the bus control register will cause the **IGNNE#** bit of the event register to be set when the **IGNNE#** signal is asserted (low).

Asserting the IGNNE# signal causes the emulator to modify its processing to ignore numeric errors, if suitably enabled to do so. The IGNNE# bit of the bus control register is then set (to 1) to cause the IGNNE# bit of the event register to be set when the IGNNE# signal is released (high).

Releasing the IGNNE# signal causes the emulator to restore the emulation to the original state. The IGNNE# bit of the bus control register is then cleared (to 0) again, to cause the IGNNE# bit of the event register to be set when the IGNNE# signal is asserted (low).

Emulator output signals

Several of the signals, BPA, BP0, IERR#, IERR#, PM1..PM0, SMLACT# are outputs that have purposes primarily defined by the needs of x86 processor emulation. They are driven from the bus control register that can be written by software.

Bus snooping

Zeus support the "Snocket" protocols for inquiry, invalidation and coherence of cache lines. The protocols are implemented in hardware and do not interrupt the processor as a result of bus activity. Cache access cycles may be "stolen" for this purpose, which may delay completion of processor memory activity.

Definition

def SnoopPhyscBus as

```
//wait for transaction on bus or inquiry cycle
do
  wait
  while BRODY# = 0
  pass 3 ← A31 3
  op ← W/R# ? W R
  cc ← CACHE# || PWT || PCD
enddef
```

Locked cycles

Locked cycles occur as a result of synchronization operations (Store swap instructions) performed by the processor. For x86 emulation, locked cycles also occur as a result of setting specific memory mapped control registers.

Locked synchronization instruction

Bus lock (LOCK#) is asserted (low) automatically as a result of store-swap instructions that generate bus activity, which always perform locked read-modify-write cycles on 64 bits of data. Note that store-swap instructions that are performed on cache sub-blocks that are in the E or M state need not generate bus activity.

Locked sequences of bus transactions

Bus lock (LOCK#) is also asserted (low) on subsequent bus transactions by writing a one (1) to the bus lock bit of the bus control register. Split cycle (SCYC) is similarly asserted (high) if a one (1) is also written to the split cycle bit of the bus emulation control register.

All subsequent bus transactions will be performed as a locked sequence of transactions, asserting bus lock (LOCK# low) and optionally split cycle (SCYC high), until zeros (0) are written to the bus lock and split cycle bits of the bus control register. The next bus transaction completes the locked sequence, releasing bus lock (LOCK# high) and split cycle (SCYC low) at the end of the transaction. If the locked transaction must be aborted because of bus activity such as backoff, a lock broken event is signalled and the bus lock is released.

Unless special care is taken, the bus transactions of all threads occur as part of the locked sequence of transactions. Software can do so by interrupting all other threads until the locked sequence is completed. Software should also take care to avoid fetching instructions during the locked sequence, such as by executing instructions out of cache or ROM memory. Software should also take care to avoid terminating the sequence with event handling prior to releasing the bus lock, such as by executing the sequence with events disabled (other than the lock broken event).

The purpose of this facility is primarily for x86 emulation purposes, in which we are willing to perform vile acts (such as stopping all the other threads) in the name of compatibility. It is possible to take special care in hardware to sort out the activity of other threads, and break the lock in response to events. In doing so, the bus unit must defer bus activity generated by other threads until the locked sequence is completed. The bus unit should inhibit event handling while the bus is locked.

Sampled at Reset

Certain pins are sampled at reset and made available in the event register.

CPU_TYP Primary or Dual processor

PICD0[DPEN#] Dual processing enable

FLUSH# Tristate test mode

INIT Built-in self test

Sampled per Clock

Certain pins are sampled per clock and changes are made available in the event register.

A20M# address bit 20 mask

BF[1:0] bus frequency

BUSCHK# bus check

FLUSH# cache flush request

FRCMC#functional redundancy check - not implemented on Pentium MMX

IGNNE#ignore numeric error

INIT re-initialize pentium processor

INTR external interrupt

NMI non-maskable interrupt

R/S# run/stop

SMI# system management

STPCLK# stop clock

Bus Access

The "Socket 7" bus performs transfers of 1-8 bytes within an octet boundary or 32 bytes on a triplet boundary.

Transfers sized at 16 bytes (hexlet) are not available as a single transaction, they are performed as two bus transactions.

Bus transactions begin by gaining control of the bus (TODO: not shown), and in the initial cycle, asserting ADS#, M/IO#, A, BE#, W/R#, CACHE#, PWT, and PCD. These signals indicate the type, size, and address of the transaction. One or more octets of data are returned on a read (the external system asserts BRDY# and/or NA# and D), or accepted on a write (TODO not shown).

The external system is permitted to affect the cacheability and exclusivity of data returned to the processor, using the KEN# and WB/WT# signals.

Definition

```
def dataLen ← AccessPhysicalBuspa.size.cc.op.wd as
  // divide transfers sized between octet and hexlet into two parts
  // also divide transfers which cross octet boundary into two parts
  if (size<128) or (size<64) and (size-8*pa2 > 64) then
    data0Len ← AccessPhysicalBuspa.64-8*pa2 0.cc.op.wd
    if cen=0 then
      pa1 ← pa2 41111103
      data1Len ← AccessPhysicalBuspa1.size-8*pa2 0-64.cc.op.wd
      data ← data1127.64 || data063 0
    end if
  end if
```

```

else
  ADS# ← 0
  M/IO# ← 1
  A31_3 ← p31_3
  for i ← 0 to 7
    BE# ← p32_0 ≤ i < p32_0*224/8
  endfor
  W#/R# ← (op = W)
  if (op=R) then
    CACHE# ← ~(cc ≥ WT)
    PWT ← (cc = WT)
    PCD ← (cc ≤ CD)
    do
      wait
      while (BROFF = 1) and (PA# = 1)
        //trict spec doesn't say whether KEN# should be ignored if no CACHE#
        //AMD spec says KEN# should be ignored if no CACHE#
        cen ← ~KEN# and (cc ≥ WT) //cen=1 if trict is cacheable
        xen ← WB/WT# and (cc = WT) //xen=1 if trict is exclusive
        if cen then
          os ← 64*p34_3
          data63-os.os ← D63.0
          do
            wait
            while BROFF = 1
              data63-64*os.os ← D63.0
            do
              wait
              while BROFF = 1
                data63-128*os.os-128*os ← D63.0
              do
                wait
                while BROFF = 1
                  data63-192*os.os-192*os ← D63.0
                end
              end
            end
          end
          os ← 64*p34_3
          data63-os.os ← D63.0
        end
      end
    end
  else
    CACHE# ← ~(tsize = 256)
    PWT ← (cc = WT)
    PCD ← (cc ≤ CD)
    do
      wait
      while (BROFF = 1) and (PA# = 1)
        xen ← WB/WT# and (cc = WT)
      end
    end
  end
  flags ← cen || xen
enddo

```

Other bus cycles

Input/Output transfers, Interrupt acknowledge, and special bus cycles (stop grant, flush acknowledge, writeback, halt, flush, shutdown) are performed by uncached loads and stores to a memory-mapped control region.

M/IO#	D/C#	W/R#	CACHE#	KEN#	cycle
0	0	0	1	x	interrupt acknowledge
0	0	1	1	x	special cycles (intel pg 6-33)
0	1	0	1	x	I/O read, 32-bits or less, non-cacheable, 16-bit address
0	1	1	1	x	I/O write, 32-bits or less, non-cacheable, 16-bit address
1	0	x	x	x	code read (not implemented)
1	1	0	1	x	non-cacheable read
1	1	0	x	1	non-cacheable read
1	1	0	0	0	cacheable read
1	1	1	1	x	non-cacheable write
1	1	1	0	x	cache writeback

Special cycles

An interrupt acknowledge cycle is performed by two byte loads to the control space ($dc=1$), the first with a byte address (ba) of 4 ($A31..3=0$, $BF4\#=0$, $BF7..5,3..0\#=1$), the second with a byte address (ba) of 0 ($A31..3=0$, $BF0\#=0$, $BF7..1\#=1$). The first byte read is ignored; the second byte contains the interrupt vector. The external system normally releases $INTR$ between the first and second byte load.

A shutdown special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of 0 ($A31..3=0$, $BF0\#=0$, $BF7..1\#=1$).

A flush special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of 1 ($A31..3=0$, $BF1\#=0$, $BF7..2,0\#=1$).

A halt special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of 2 ($A31..3=0$, $BF2\#=0$, $BF7..3,1..0\#=1$).

A stop grant special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of $0x12$ ($A31..3=2$, $BF2\#=0$, $BF7..3,1..0\#=1$).

A writeback special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of 3 ($A31..3=0$, $BF3\#=0$, $BF7..4,2..0\#=1$).

A flush acknowledge special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of 4 ($A31..3=0$, $BF4\#=0$, $BF7..5,3..0\#=1$).

A back trace message special cycle is performed by a byte store to the control space ($dc=1$) with a byte address (ba) of 5 ($A31..3=0$, $BF5\# = 0$, $BF7..6,4..0\# = 1$).

Performing load or store operations of other sizes (doublet, quadlet, octlet, hexlet) to the control space ($dc=1$) or operations with other byte address (ba) values produce bus operations which are not defined by the "Super Socket 7" specifications and have undefined effect on the system.

I/O cycles

An input cycle is performed by a byte, doublet, or quadlet load to the data space ($dc=0$), with a byte address (ba) of the I/O address. The address may not be aligned, and if it crosses an octlet boundary, will be performed as two separate cycles.

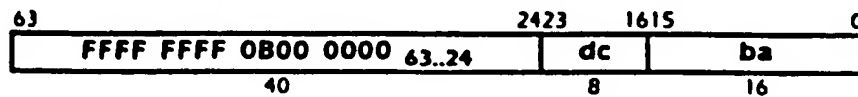
An output cycle is performed by a byte, doublet, or quadlet store to the data space ($dc=0$), with a byte address (ba) of the I/O address. The address may not be aligned, and if it crosses an octlet boundary, will be performed as two separate cycles.

Performing load or store operations of other sizes (octlet, hexlet) to the data space ($dc=0$) produce bus operations which are not defined by the "Super Socket 7" specifications and have undefined effect on the system.

Physical address

The other bus cycles are accessed explicitly by uncached memory accesses to particular physical address ranges. Appropriately sized load and store operations must be used to perform the specific bus cycles required for proper operations. The dc field must equal 0 for I/O operations, and must equal 1 for control operations. Within this address range, bus transactions are sized no greater than 4 bytes (quadlet) and do not cross quadlet boundaries.

The physical address of a other bus cycle data/control dc , byte address ba is:



Definition

def data ← AccessPhysicaOtherBus(pa,size,op,wd) as

```
// divide transfers sized between octlet and hexlet into two parts
// also divide transfers which cross octlet boundary into two parts
if (64 < size < 128) or ((size < 64) and (size + 8 * pa2_0 > 64)) then
    data0 ← AccessPhysicaOtherBus(pa,64-8*pa2_0,op,wd)
    pa1 ← pa63..41111103
    data1 ← AccessPhysicaOtherBus(pa1,size+8*pa2_0-64,op,wd)
    data ← data1127..64 || data063..0
else
    ADS# ← 0
    M/IO# ← 0
```

```
DVCE ← pa16
A31:3 ← 016 || pa15:3
for i ← 0 to 7
    BEp ← pa2.0 ≤ i < pa2.0*size/8
endfor
W/Rs ← {op = W}
CACHEs ← 1
PWT ← 1
PCD ← 1
do
    wait
    while (BRODs = 1) and (NAs = 1)
    if {op=R} then
        os ← 64*pa3
        data63-os.os ← D63.0
    endif
enddo
enddo
```


Events and Threads

Exceptions signal several kinds of events: (1) events that are indicative of failure of the software or hardware, such as arithmetic overflow or parity error, (2) events that are hidden from the virtual process model, such as translation buffer misses, (3) events that infrequently occur, but may require corrective action, such as floating-point underflow. In addition, there are (4) external events that cause scheduling of a computational process, such as clock events or completion of a disk transfer.

Each of these types of events require the interruption of the current flow of execution, handling of the exception or event, and in some cases, descheduling of the current task and rescheduling of another. The Zeus processor provides a mechanism that is based on the multi-threaded execution model of Mach. Mach divides the well-known UNIX process model into two parts, one called a task, which encompasses the virtual memory space, file and resource state, and the other called a thread, which includes the program counter, stack space, and other register file state. The sum of a Mach task and a Mach thread exactly equals one UNIX process, and the Mach model allows a task to be associated with several threads. On one processor at any one moment in time, at least one task with one thread is running.

In the taxonomy of events described above, the cause of the event may either be synchronous to the currently running thread, generally types 1, 2, and 3, or asynchronous and associated with another task and thread that is not currently running, generally type 4.

For these events, Zeus will suspend the currently running thread in the current task, saving a minimum of registers, and continue execution at a new program counter. The event handler may perform some minimal computation and return, restoring the current threads' registers, or save the remaining registers and switch to a new task or thread context.

Facilities of the exception, memory management, and interface systems are themselves memory mapped, in order to provide for the manipulation of these facilities by high-level language, compiled code. The sole exception is the register file itself, for which standard store and load instructions can save and restore the state.

Definition

```
def Thread(t) as
  forever
    catch exception
      if (EventRegister and EventMask[t]) ≠ 0 then
        if ExceptionState=0 then
          raise EventInterrupt
        endif
      endif
      inst ← LoadMemoryX(ProgramCounter, ProgramCounter, 32, 1)
      Instruction(inst)
    endcatch
  case exception of
    EventInterrupt,
    ReservedInstruction,
    AccessDisallowedByVirtualAddress,
    AccessDisallowedByTag.
```

```

AccessDisallowedByGlobalTB,
AccessDisallowedByLocalTB,
AccessDetailRequiredByTag,
AccessDetailRequiredByGlobalTB,
AccessDetailRequiredByLocalTB,
MissInGlobalTB,
MissInLocalTB,
FixedPointArithmetic,
FloatingPointArithmetic,
GatewayDisallowed:
  case ExceptionState of
    0:
      PerformException(exception)
    1:
      PerformException(SecondException)
    2:
      raise ThirdException
  endcase
TakenBranch:
  ContinuationState ← (ExceptionState=0) ? 0 : ContinuationState
TakenBranchContinue:
  /* nothing */
/* one, others:
  ProgramCounter ← ProgramCounter + 4
  ContinuationState ← (ExceptionState=0) ? 0 : ContinuationState
endcase
endforever
enddel

```

Definition

```

del PerformException(exception) as
  v ← (exception > 7) ? 7 : exception
  l ← LoadMemory(ExceptionBase, ExceptionBase+Thread*128+64*8*v, 64, L)
  if ExceptionState = 0 then
    u ← RegRead(3, 128) || RegRead(2, 128) || RegRead(1, 128) || RegRead(0, 128)
    StoreMemory(ExceptionBase, ExceptionBase+Thread*128, 512, L, u)
    RegWrite(0, 64, ProgramCounter_63..2 || PrivilegeLevel)
    RegWrite(1, 64, ExceptionBase+Thread*128)
    RegWrite(2, 64, exception)
    RegWrite(3, 64, FailingAddress)
  endif
  PrivilegeLevel ← 1..0
  ProgramCounter ← 63..2 || 02
  case exception of
    AccessDetailRequiredByTag,
    AccessDetailRequiredByGlobalTB,
    AccessDetailRequiredByLocalTB:
      ContinuationState ← ContinuationState + 1
    others:
      /* nothing */
  endcase
  ExceptionState ← ExceptionState + 1
enddel

```

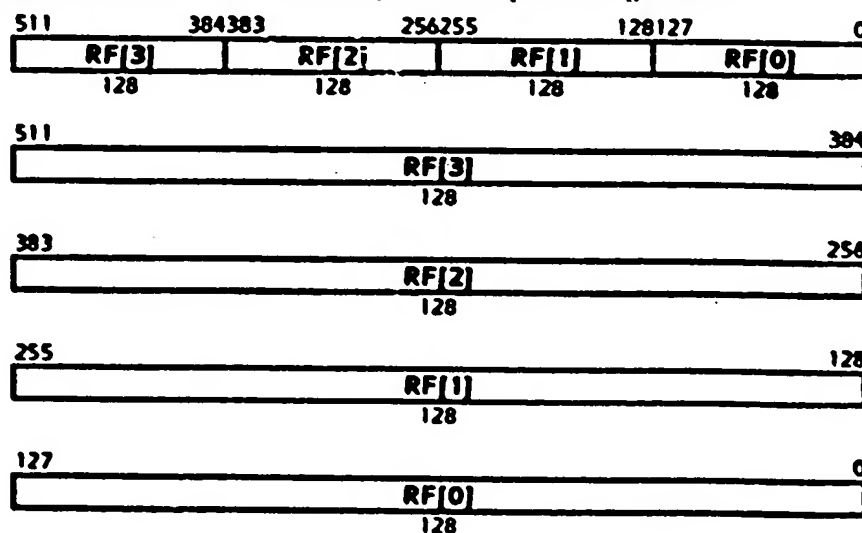
Definition

```
def PerformAccessDetail(exception) as
  if (ContinuationState = 0) or (ExceptionState = 0) then
    raise exception
  else
    ContinuationState ← ContinuationState - 1
  endif
enddef
```

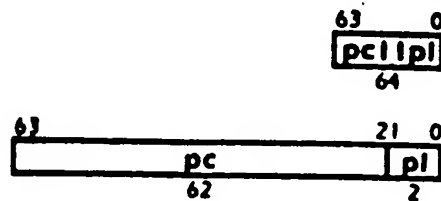
Definition

```
def BranchBack(rd,rc,rb) as
  c ← RegRead(rc, 64)
  if (rd = 0) or (rc = 0) or (rb = 0) then
    raise ReservedInstruction
  endif
  a ← LoadMemory(ExceptionBase, ExceptionBase + Thread * 128, 128, 1)
  if PrivilegeLevel > c1_0 then
    PrivilegeLevel ← c1_0
  endif
  ProgramCounter ← c63_2 || 02
  ExceptionState ← 0
  RegWrite(rd, 128, a)
  raise TakenBranchContinue
enddef
```

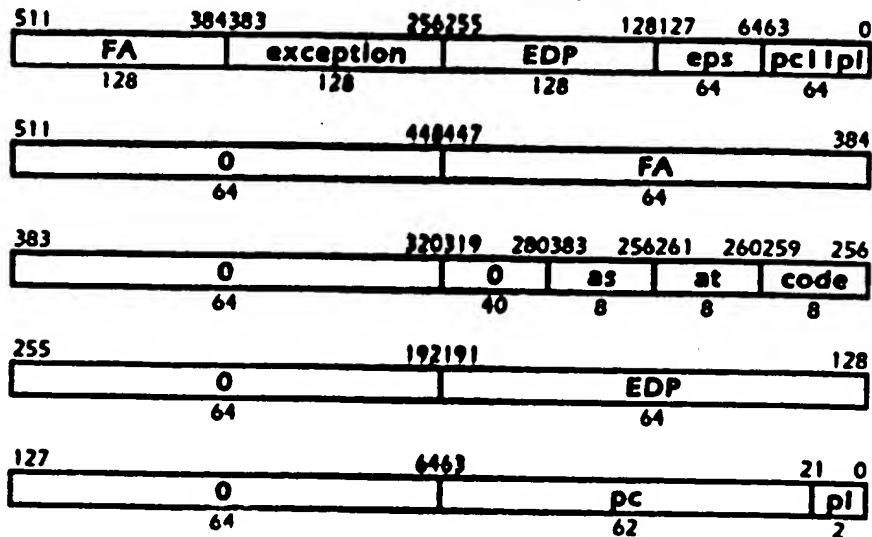
The following data is stored into memory at the Exception Storage Address



The following data is loaded from memory at the Exception Vector Address:



The following data replaces the original contents of R[3..0]:



at: access type: 0=r, 1=w, 2=x, 3=h

as: access size in bytes

TODC: add size, access type to exception data in pseudocode.

Ephemeral Program State

Ephemeral Program State (EPS) is defined as program state which affects the operation of certain instructions, but which does not need to be saved and restored as part of user state.

Because these bits are not saved and restored, the sizes and values described here are not visible to software. The sizes and values described here were chosen to be convenient for the definitions in this documentation. Any mapping of these values which does not alter the functions described may be used in a conforming implementation. For example, either of the EPS states may be implemented as a thermometer-coded vector, or the ContinuationState field may be represented with specific values for each AccessDetailRequired exception which an instruction execution may encounter.

There are eight bits of FPS:

bits	Name	Meaning
1..0	ExceptionState	<p>0: Normal processing. Asynchronous events and Synchronous exceptions enabled.</p> <p>1: Event/Exception handling: Synchronous exceptions cause SecondException. Asynchronous events are masked.</p> <p>2: Second exception handling: Synchronous exceptions cause a machine check. Asynchronous events are masked.</p> <p>3: Illegal state</p> <p>This field is incremented by handling an event or exception, and cleared by the Branch Back Instruction.</p>
7..2	ContinuationState	<p>Continuation state for AccessDetailRequired exceptions. A value of zero enables all exceptions of this kind. The value is increased by one for each AccessDetailRequired exception handled, for which that many AccessDetailRequired exceptions are continued past (ignored) on re-execution in normal processing (ex=0). Any other kind of exception, or the completion of an instruction under normal processing causes the continuation state to be reset to zero. State does not need to be saved on context switch.</p>

The ContinuationState bits are ephemeral because if they are cleared as a result of a context switch, the associated exceptions can happen over again. The AccessDetail exception handlers will then set the bits again, as they were before the context switch. In the case where an AccessDetail exception handler must indicate an error, care must be taken to perform some instruction at the target of the Branch Back instruction by the exception handler is exited that will operate properly with ContinuationState#0.

The ExceptionState bits are ephemeral because they are explicitly set by event handling and cleared by the termination of event handling, including event handling that results in a context switch.

Event Register

Events are single-bit messages used to communicate the occurrence of events between threads and interface devices.

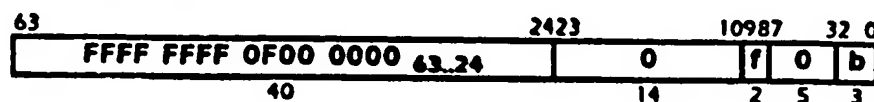


The Event Register appears at several locations in memory, with slightly different side effects on read and write operations.

offset	side effect on read	side effect on write
0	none: return event register contents	normal: write data into event register
256	stall thread until contents of event register is non-zero, then return event register contents	stall thread until bitwise and of data and event register contents is non-zero
512	return zero value (so read-modify-write for byte/doublet/quadlet store works)	one bits in data set (to one) corresponding event register bits
768	return zero value (so read-modify-write for byte/doublet/quadlet store works)	one bits in data clear (to zero) corresponding event register bits

Physical address

The Event Register appears at three different locations, for which three functions of the Event Register are performed as described above. The physical address of an Event Register for function *f*, byte *b* is:



Definition

```

def data ← AccessPhysicalEventRegister(pa, op, wdata) as
  f ← pa7_8
  if (pa23_10 = 0) and (pa7_4 = 0) and (f = 1) then
    case f || op of
      0 || R:
        data ← 064 || EventRegister
      2 || R, 3 || R:
        data ← 0
      0 || W:
        EventRegister ← wdata63_0
      2 || W:
        EventRegister ← EventRegister or wdata63_0
      3 || W:
        EventRegister ← EventRegister and ~wdata63_0
    endcase
  else
    data ← 0
  endif
enddef

```

Events:

The table below shows the events and their corresponding event number. The priority of these events is soft, in that dispatching from the event register is controlled by software.

TODO notwithstanding the above, using the `E.LOGMOST.U` instruction is handy for prioritizing these events, so if you've got a preference as to numbering, speak up!

number	event
0	Clock
1	A20M# active
2	BF0 active
3	BF1 active
4	BF2 active
5	BUSCHK# active
6	FLUSH# active
7	FRCMC# active
8	IGNNE# active
9	INIT active
10	INTR active
11	NMI active
12	SMI# active
13	STPCLK# active
14	CPUTYP active at reset (Primary vs Dual processor)
15	DPEN# active at reset (Dual processing enable - driven low by dual processor)
16	FLUSH# active at reset (tristate test mode)
17	INIT active at reset
18	Bus lock broken
19	BRYRC# active at reset (drive strength)
20	

Event Mask

The Event Mask (one per thread) control whether each of the events described above is permitted to cause an exception in the corresponding thread.

Physical address

There are as many Event Masks as threads. The physical address of an Event Mask for thread `th`, byte `b` is:

Definition

```

def data ← AccessPhysicalEventMask(pa.op.wdata) as
  th ← pa23..19
  if (th < T) and (pa18..4 = 0) then
    case op of
      R:
        data ← 064 || EventMask[th]
      W:
        EventMask[th] ← wdata63..0
    endcase
  else
    data ← 0
  endif
enddef

```


Exceptions:

The table below shows the exceptions, the corresponding exception number, and the parameter supplied by the exception handler in register 3.

number	exception	parameter (register 3)
0	EventInterrupt	
1	MissInGlobalTB	global address
2	AccessDetailRequiredByTag	global address
3	AccessDetailRequiredByGlobalTB	global address
4	AccessDetailRequiredByLocalTB	local address
5		
6	SecondException	
7	ReservedInstruction	instruction
8	AccessDisallowedByVirtualAddress	local address
9	AccessDisallowedByTag	global address
10	AccessDisallowedByGlobalTB	global address
11	AccessDisallowedByLocalTB	local address
12	MissInLocalTB	local address
13	FixedPointArithmetic	instruction
14	FloatingPointArithmetic	instruction
15	GatewayDisallowed	none
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
	TakenBranch	
	TakenBranchContinue	

GlobalTB Miss Handler

The GlobalTB Miss exception occurs when a load, store, or instruction fetch is attempted while none of the GlobalTB entries contain a matching virtual address. The Zeus processor uses a fast software-based exception handler to fill in a missing GlobalTB entry.

There are several possible ways that software may maintain page tables. For purposes of this discussion, it is assumed that a virtual page table is maintained, in which 128 bit GTB values for each 4k byte page in a linear table which is itself in virtual memory. By maintaining the page table in virtual memory, very large virtual spaces may be managed without keeping a large amount of physical memory dedicated to page tables.

Because the page table is kept in virtual memory, it is possible that a valid reference may cause a second GTBMiss exception if the virtual address that contains the page table is not present in the GTB. The processor is designed to permit a second exception to occur within an exception handler, causing a branch to the SecondException handler. However, to simplify the hardware involved, a SecondException exception saves no specific information about the exception - handling depends on keeping enough relevant information in registers to recover from the second exception.

Zeus is a multithreaded processor, which creates some special considerations in the exception handler. Unlike a single-threaded processor, it is possible that multiple threads may nearly simultaneously reference the same page and invoke two or more GTB misses, and the fully-associative construction of the GTB requires that there be no more than one matching entry for each global virtual address. Zeus provides a search-and-insert operation (GTBUpdateFill) to simplify the handling of the GTB. This operation also uses hardware GTB pointer registers to select GTB entries for replacement in FIFO priority.

A further problem is that software may need to modify the protection information contained in the GTB, such as to remove read and/or write access to a page in order to infer which parts of memory are in use, or to remove pages from a task. These modifications may occur concurrently with the GTBMiss handler, so software must take care to properly synchronize these operations. Zeus provides a search-and-update operation (GTBUpdate) to simplify updating GTB entries.

When a large number of page table entries must be changed, noting the limited capacity of the GTB can reduce the work. Reading the GTB can be less work than matching all modified entries against the GTB contents. To facilitate this, Zeus also provides read access to the hardware GTB pointers to further permit scanning the GTB for entries which have been replaced since a previous scan. GTB pointer wraparound is also logged, so it can be determined that the entire GTB needs to be scanned if all entries have been replaced since a previous scan.

In the code below, offsets from r1 are used with the following data structure

Offset	Meaning
0..15	r0 save
16..32	r1 save
32..47	r2 save
48..63	r3 save
512..527	r4 save
528..535	BasePT
536..543	GTBUpdateFill
544..559	DummyPT
560..639	available 96 bytes

BasePT = 512 + 16

GTBUpdateFill = BasePT + 8

DummyPT = GTBUpdateFill + 8

On a GTBMiss, the handler retrieves a base address for the virtual page table and constructs an index by shifting away the page offset bits of the virtual address. A single 128-bit indexed load retrieves the new GTB entry directly (except that a virtual page table miss causes a second exception, handled below). A single 128-bit store to the GTBUpdateFill location places the entry into the GTB, after checking to ensure that a concurrent handler has not already placed the entry into the GTB.

Code for GlobalTBMiss:

```

        h64la      r2=r1,BasePT          //base address for page table
        ashn       r3@12                //4k pages
        h128la     r3=r2,r3             //retrieve page table, SecExc if bad va
2.       h64la     r2=r1,GTBUpdateFill    //pointer to GTB update location
        s128la     r3,r2,0              //save new TB entry
        h128la     r3=r1,48             //restore r3
        h128la     r2=r1,32             //restore r2
        h128la     r1=r1,16             //restore r1
        bback      //restore r0 and return
    
```

A second exception occurs on a virtual page table miss. It is possible to service such a page table miss directly, however, the page offset bits of the virtual address have been shifted away, and have been lost. These bits can be recovered: in such a case, a dummy GTB entry is constructed, which will cause an exception other than GTBMiss upon returning. A re-execution of the offending code will then invoke a more extensive handler, making the full virtual address available.

For purposes of this example, it is assumed that checking the contents of r2 against the contents of BasePT is a good way to ensure that the second exception handler was entered from the GlobalTBMiss handler.

Code for SecondException:

```

        s128la     r4,r1,512            //save r4
        h64la     r4=r1,BasePT          //base address for page table
        bne       r2,r4,1f             //did we lose at page table load?
    
```

```

01201a      r2=r1.DummyPT      //dummy page table, shifted left 64-12 bits
01201a      r3=r2.64+12       //combine page number with dummy entry
01201a      r4=r1.512         //restore r4
0           2b                //fall back into GTB Miss handler

```

Exceptions in detail

There are no special registers to indicate details about the exception, such as the virtual address at which an access was attempted, or the operands of a floating-point operation that results in an exception. Instead, this information is available via general-purpose registers or registers saved in memory.

When a synchronous exception or asynchronous event occurs, the original contents of registers 0..3 are saved in memory and replaced with (1) program counter, privilege level, and ephemeral program state, (2) exception code, and (3) when applicable, failing address or instruction. A new program counter and privilege level is loaded from memory and execution begins at the new address. After handling the exception and restoring all but one register, a branch-back instruction restores the final register and resumes execution.

During exception handling, any asynchronous events are kept pending until a BranchBack instruction is performed. By this mechanism, we can handle exceptions and events one at a time, without the need to interrupt and stack exceptions. Software should take care to avoid keeping the handling of asynchronous events pending for too long.

When a second exception occurs in a thread which is handling an exception, all the above operations occur, except for the saving and replacing of registers 0..3 in memory. A distinct exception code SecondException replaces the normal exception code. By this mechanism, a fast exception handler for GlobalTBMiss can be written, in which a second GlobalTBMiss or FixedPointOverflow exception may safely occur.

When a third exception occurs in a thread which is handling an exception, an immediate transfer of control occurs to the machine check vector address, with information about the exception available in the machine check cause field of the status register. The transfer of control may overwrite state that may be necessary to recover from the exception; the intent is to provide a satisfactory post-mortem indication of the characteristics of the failure.

This section describes in detail the conditions under which exceptions occur, the parameters passed to the exception handler, and the handling of the result of the procedure.

Reserved Instruction

The ReservedInstruction exception occurs when an instruction code which is reserved for future definition as part of the Zeus architecture is executed.

Register 3 contains the 32 bit instruction.

Access Disallowed by virtual address

This exception occurs when a load, store, branch, or gateway refers to an aligned memory operand with an improperly aligned address, or if architecture description parameter LB=1, may also occur if the add or increment of the base register or program counter which generates the address changes the unmasked upper 16 bits of the local address.

Register 3 contains the local address to which the access was attempted.

Access disallowed by tag

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching cache tag entry does not permit this access.

Register 3 contains the global address to which the access was attempted.

Access detail required by tag

This exception occurs when a read (load), write (store), or execute attempts to access a virtual address for which the matching virtual cache entry would permit this access, but the detail bit is set.

Register 3 contains the global address to which the access was attempted.

Description

The exception handler should determine accessibility. If the access should be allowed, the continuepastdetail bit is set and execution returns. Upon return, execution is restarted and the access will be retried. Even if the detail bit is set in the matching virtual cache entry, access will be permitted.

Access disallowed by global TB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching global TB entry does not permit this access.

Register 3 contains the global address to which the access was attempted.

Description

The exception handler should determine accessibility, modify the virtual memory state if desired, and return if the access should be allowed. Upon return, execution is restarted and the access will be retried.

Access detail required by global TB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching global TB entry would permit this access, but the detail bit in the global TB entry is set.

Register 3 contains the global address to which the access was attempted.

Description

The exception handler should determine accessibility and return if the access should be allowed. Upon return, execution is restarted and the access will be allowed. If the access is not to be allowed, the handler should not return.

Global TB miss

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which no global TB entry matches.

Register 3 contains the global address to which the access was attempted.

Description

The exception handler should load a global TB entry that defines the translation and protection for this address. Upon return, execution is restarted and the global TB access will be attempted again.

Access disallowed by local TB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching local TB entry does not permit this access.

Register 3 contains the local address to which the access was attempted.

Description

The exception handler should determine accessibility, modify the virtual memory state if desired, and return if the access should be allowed. Upon return, execution is restarted and the access will be retried.

Access detail required by local TB

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which the matching local TB entry would permit this access, but the detail bit in the local TB entry is set.

Register 3 contains the local address to which the access was attempted.

Description

The exception handler should determine accessibility and return if the access should be allowed. Upon return, execution is restarted and the access will be allowed. If the access is not to be allowed, the handler should not return.

Local TB miss

This exception occurs when a read (load), write (store), execute, or gateway attempts to access a virtual address for which no local TB entry matches.

Register 3 contains the local address to which the access was attempted.

Description

The exception handler should load a local TB entry that defines the translation and protection for this address. Upon return, execution is restarted and the local TB access will be attempted again.

Floating-point arithmetic

Register 3 contains the 52-bit instruction.

Description

The address of the instruction that was the cause of the exception is passed as the contents of register 0. The exception handler should attempt to perform the function specified in the instruction and service any exceptional conditions that occur.

Fixed-point arithmetic

Register 3 contains the 32-bit instruction.

Description

The address of the instruction which was the cause of the exception is passed as the contents of register 0. The exception handler should attempt to perform the function specified in the instruction and service any exceptional conditions that occur.

Reset and Error Recovery

Certain external and internal events cause the processor to invoke reset or error recovery operations. These operations consist of a full or partial reset of critical machine state, including initialization of the threads to begin fetching instructions from the start vector address. Software may determine the nature of the reset or error by reading the value of the control register, in which finding the reset bit set (1) indicates that a reset has occurred, and finding both the reset bit cleared (0) indicates that a machine check has occurred. When either a reset or machine check has been indicated, the contents of the status register contain more detailed information on the cause.

Definition

```
def PerformMachineCheck(cause) as  
  ResetVirtualMemory()  
  ProgramCounter ← StartVectorAddress  
  PrivilegeLevel ← 3  
  StatusRegister ← cause  
enddef
```

Reset

A reset may be caused by a power on reset, a bus reset, a write of the control register which sets the reset bit, or internally detected errors including meltdown detection, and double check.

A reset causes the processor to set the configuration to minimum power and low clock speed, note the cause of the reset in the status register, stabilize the phase locked loops, disable the MMU from the control register, and initialize all threads to begin execution at the start vector address.

Other system state is left undefined by reset and must be explicitly initialized by software; this explicitly includes the thread register state, LTB and GTB state, superspring state, and external interface devices. The code at the start vector address is responsible for initializing these remaining system facilities, and reading further bootstrap code from an external ROM.

Power-on Reset

A reset occurs upon initial power on. The cause of the reset is noted by initializing the Status Register and other registers to the reset values noted below.

Bus Reset

A reset occurs upon observing that the RESET signal has been at active. The cause of the reset is noted by initializing the Status Register and other registers to the reset values noted below.

Control Register Reset

A reset occurs upon writing a one to the reset bit of the Control Register. The cause of the reset is noted by initializing the Status Register and other registers to the reset values noted below.

Meltdown Detected Reset

A reset occurs if the temperature is above the threshold set by the meltdown margin field of the configuration register. The cause of the reset is noted by setting the meltdown detected bit of the Status Register.

Double Check Reset

A reset occurs if a second machine check occurs that prevents recovery from the first machine check. Specifically, the occurrence of an exception in event thread, watchdog timer error, or bus error while any machine check cause bit is still set in the Status Register results in a double machine check reset. The cause of the reset is noted by setting the double check bit of the Status Register.

Machine Check

Detected hardware errors, such as communications errors in the bus, a watchdog timeout error, or internal cache parity errors, invoke a machine check. A machine check will disable the MMU, to translate all local virtual addresses to equal physical addresses, note the cause of the exception in the Status Register, and transfer control of the all threads to the start vector address. This action is similar to that of a reset, but differs in that the configuration settings, and thread state are preserved.

Recovery from machine checks depends on the severity of the error and the potential loss of information as a direct cause of the error. The start vector address is designed to reach internal ROM memory, so that operation of machine check diagnostic and recovery code need not depend on proper operation or contents of any external device. The program counter and register file state of the thread prior to the machine check is lost (except for the portion of the program counter saved in the Status Register), so diagnostic and recovery code must not assume that the register file state is indicative of the prior operating state of the thread. The state of the thread is frozen similarly to that of an exception.

Machine check diagnostic code determines the cause of the machine check from the processor's Status Register, and as required, the status and other registers of external bus devices.

Recovery code will generally consume enough time that real-time interface performance targets may have been missed. Consequently, the machine check recovery software may need to repair further damage, such as interface buffer underruns and overruns as may have occurred during the intervening time.

This final recovery code, which re-initializes the state of the interface system and recovers a functional event thread state, may return to using the complete machine resources, as the condition which caused the machine check will have been resolved.

The following table lists the causes of machine check errors.

Parity or uncorrectable error in on-chip cache Parity or communications error in system bus Event Thread exception Watchdog timer
--

machine check errors

Parity or Uncorrectable Error in Cache

When a parity or uncorrectable error occurs in an on-chip cache, such an error is generally non-recoverable. These errors are non-recoverable because the data in such caches may reside anywhere in memory, and because the data in such caches may be the only up-to-date copy of that memory contents. Consequently, the entire contents of the memory store is lost, and the severity of the error is high enough to consider such a condition to be a system failure.

The machine check provides an opportunity to report such an error before shutting down a system for repairs.

There are specific means by which a system may recover from such an error without failure, such as by restarting from a system-level checkpoint, from which a consistent memory state can be recovered.

Parity or Communications Error in Bus

When a parity or communications error occurs in the system bus, such an error may be partially recoverable.

Bits corresponding to the affected bus operation are set in the processor's Status Register. Recovery software should determine which devices are affected, by querying the Status Register of each device on the affected MediaChannel channels.

A bus timeout may result from normal self configuration activities.

If the error is simply a communications error, resetting appropriate devices and restarting tasks may recover from the error. Read and write transactions may have been underway at the time of a machine check and may or may not be reflected in the current system state.

If the error is from a parity error in memory, the contents of the affected area of memory is lost, and consequently the tasks associated with that memory must generally be aborted, or resumed from a task level checkpoint. If the contents of the affected memory can be recovered from mass storage, a complete recovery is possible.

If the affected memory is that of a critical part of the operating system, such a condition is considered a system failure, unless recovery can be accomplished from a system-level checkpoint.

Watchdog Timeout Error

A watchdog timeout error indicates a general software or hardware failure. Such an error is generally treated as non-recoverable and fatal.

Event Thread Exception

When an event thread suffers an exception, the cause of the exception and a portion of the virtual address at which the exception occurred are noted in the Status Register. Because under normal circumstances, the event thread should be designed not to encounter exceptions, such exceptions are treated as non-recoverable, fatal errors.

Reset state

A reset or machine check causes the Zeus processor to stabilize the phase locked loops, disable the local and global TB, to translate all local virtual addresses to equal physical addresses, and initialize all threads to begin execution at the start vector address.

Start Address

The start address is used to initialize the threads with a program counter upon a reset, or machine check. These causes of such initialization can be differentiated by the contents of the Status Register.

The start address is a virtual address which, when "translated" by the local TB and global TB to a physical address, is designed to access the internal ROM code. The internal ROM space is chosen to minimize the number of internal resources and interfaces that must be operated to begin execution or recover from a machine check.

Virtual/physical address	description
0xFFFF FFFF FFFF FFEC	start vector address

Definition

```

def StartProcessor as
  forever
    catch check
      EnableWatchdog ← 0
      fork RunClock
      ControlRegister62 ← 0
      for th ← 0 to T-1
        ProgramCounter[th] ← 0xFFFF FFFF FFFF FFEC
        PrivilegeLevel[th] ← 3
        fork Thread[th]
      endfor
  endfor

```

```
endcatch
kill RunClock
for th ← 0 to T-1
    kill Thread(th)
endfor
PerformMachineCheck(check)
endforever
enddef

def PerformMachineCheck(check) as
    case check of
        ClockWatchdog:
        CacheError:
        ThirdException:
    endcase
enddef
```

Internal ROM Code

Zeus internal ROM code performs reset initialization of on-chip resources, including the IZC and LXC, followed by self-testing. The BIOS ROM should be scanned for a special prefix that indicates that Zeus native code is present in the ROM, in which case the ROM code is executed directly, otherwise execution of a BIOS-level x86 emulator is begun.

Memory and Devices

Physical Memory Map

Zeus defines a 64-bit physical address, but while residing in a S7 pin-out, can address a maximum of 4Gb of main memory. In other packages the core Zeus design can provide up to 64-bit external physical address spaces. Bit 63..32 of the physical address distinguishes between internal (on-chip) physical addresses, where bits 63..32=FFFFFFFF, and external (off-chip) physical addresses, where bits 63..32=FTFFFFFFF.

Address range	bytes	Meaning
0000 0000 0000 0000.FFFF FFFF	4G	External Memory
0000 0001 0000 0000.FFFF FFFF	16K-8G	External Memory expansion
FFFF FFFF 0000 0000.FFFF FFFF 0002 0FFF	128K-4K	Level One Cache
FFFF FFFF 0002 1000.FFFF FFFF 08FF FFFF	144M-132K	Level One Cache expansion
FFFF FFFF 0900 0000.FFFF FFFF 0900 007F	128	Level One Cache redundancy
FFFF FFFF 0900 0080.FFFF FFFF 09FF FFFF	16M-128	LOC redundancy expansion
FFFF FFFF 0A00 0000- $2^{19} \times 16$	$8 \times 2^{21}E$	LTB thread t entry e
FFFF FFFF 0A00 0000.FFFF FFFF 0AFF FFFF	$8 \times 2^{21}E$	LTB max $8 \times 2^{21}E = 16M$ bytes
FFFF FFFF 0B00 0000.FFFF FFFF 0BFF FFFF	16M	Special Bus Operations
FFFF FFFF 0C00 0000- $4 \times 2^{19} \times GT \times 16$	$72 \times 4 \times GE-GT$	GTB thread t entry e
FFFF FFFF 0C00 0000.FFFF FFFF 0CFF FFFF	$72 \times 4 \times GE-GT$	GTB max $2^{5+4+15} = 16M$ bytes
FFFF FFFF 0D00 0000- $4 \times 2^{19} \times GT$	$16 \times 2^{19} \times GT$	GTBUpdate thread t
FFFF FFFF 0D00 0100- $4 \times 2^{19} \times GT$	$16 \times 2^{19} \times GT$	GTBUpdateFill thread t
FFFF FFFF 0D00 0200- $4 \times 2^{19} \times GT$	$8 \times 2^{19} \times GT$	GTBLast thread t
FFFF FFFF 0D00 0300- $4 \times 2^{19} \times GT$	$8 \times 2^{19} \times GT$	GTBFirst thread t
FFFF FFFF 0D00 0400- $4 \times 2^{19} \times GT$	$8 \times 2^{19} \times GT$	GTBBump thread t
FFFF FFFF 0E00 0000- 2^{19}	8T	Event Mask thread t
FFFF FFFF 0F00 0000.FFFF FFFF 0F00 00FF	256-8	Reserved
FFFF FFFF 0F00 0100.FFFF FFFF 0F00 0107	8	Event Register with stall
FFFF FFFF 0F00 0108.FFFF FFFF 0F00 01FF	256-8	Reserved
FFFF FFFF 0F00 0200.FFFF FFFF 0F00 0207	8	Event Register bit set
FFFF FFFF 0F00 0208.FFFF FFFF 0F00 02FF	256-8	Reserved
FFFF FFFF 0F00 0300.FFFF FFFF 0F00 0307	8	Event Register bit clear
FFFF FFFF 0F00 0308.FFFF FFFF 0F00 03FF	256-8	Reserved
FFFF FFFF 0F00 0400.FFFF FFFF 0F00 0407	8	Clock Cycle
FFFF FFFF 0F00 0408.FFFF FFFF 0F00 04FF	256-8	Reserved
FFFF FFFF 0F00 0500.FFFF FFFF 0F00 0507	8	Thread
FFFF FFFF 0F00 0508.FFFF FFFF 0F00 05FF	256-8	Reserved
FFFF FFFF 0F00 0600.FFFF FFFF 0F00 0607	8	Clock Event
FFFF FFFF 0F00 0608.FFFF FFFF 0F00 06FF	256-8	Reserved
FFFF FFFF 0F00 0700.FFFF FFFF 0F00 0707	8	Clock Watchdog
FFFF FFFF 0F00 0708.FFFF FFFF 0F00 07FF	256-8	Reserved
FFFF FFFF 0F00 0800.FFFF FFFF 0F00 0807	8	Tally Counter 0
FFFF FFFF 0F00 0808.FFFF FFFF 0F00 08FF	256-8	Reserved
FFFF FFFF 0F00 0900.FFFF FFFF 0F00 0907	8	Tally Control 0
FFFF FFFF 0F00 0908.FFFF FFFF 0F00 09FF	256-8	Reserved
FFFF FFFF 0F00 0A00.FFFF FFFF 0F00 0A07	8	Tally Counter 1

FFFF FFFF 0F00 0A08..FFFF FFFF 0F00 0AFF	256-8	Reserved
FFFF FFFF 0F00 0B00..FFFF FFFF 0F00 0B07	8	Tally Control 1
FFFF FFFF 0F00 0B08..FFFF FFFF 0F00 0BFF	256-8	Reserved
FFFF FFFF 0F00 0C00..FFFF FFFF 0F00 0C07	8	Exception Base
FFFF FFFF 0F00 0C08..FFFF FFFF 0F00 0CFF	512-8	Reserved
FFFF FFFF 0F00 0C00..FFFF FFFF 0F00 0D07	8	Bus Control Register
FFFF FFFF 0F00 0D08..FFFF FFFF 0F00 0DFF	512-8	Reserved
FFFF FFFF 0F00 0E00..FFFF FFFF 0F00 0E07	8	Status Register
FFFF FFFF 0F00 0208..FFFF FFFF 0F00 02FF	256-8	Reserved
FFFF FFFF 0F00 0F00..FFFF FFFF 0F00 0F07	8	Control Register
FFFF FFFF 0F00 0F08..FFFF FFFF FEFF FFFF		Reserved
FFFF FFFF FF00 0000..FFFF FFFF FFFE FFFF	16M-64k	Internal ROM expansion
FFFF FFFF FFFF 0000..FFFF FFFF FFFF FFFF	64K	Internal ROM

The suffixes in the table above have the following meanings:

letter	name	2x	"binary"	10 ^x	"decimal"
b	bits				
B	bytes	0	1	0	1
K	kilo	10	1 024	3	1 000
M	mega	20	1 048 576	6	1 000 000
G	giga	30	1 073 741 824	9	1 000 000 000
T	tera	40	1 099 511 627 776	12	1 000 000 000 000
P	peta	50	1 125 899 906 842 624	15	1 000 000 000 000 000
E	exa	60	1 152 921 504 606 846 976	18	1 000 000 000 000 000 000

Definition

```

def data ← ReadPhysical(pa,size) as
    data,flags ← AccessPhysical(pa,size,W,A,R,O)
enddef

def WritePhysical(pa,size,wdata) as
    data,flags ← AccessPhysical(pa,size,W,A,W,wdata)
enddef

def data,flags ← AccessPhysical(pa,size,cc,op,wdata) as
    if (0x0000000000000000 ≤ pa ≤ 0x00000000FFFFFFFF) then
        data,flags ← AccessPhysicalBus(pa,size,cc,op,wdata)
    else
        data ← AccessPhysicalDevices(pa,size,op,wdata)
        flags ← 1
    endif
enddef

def data ← AccessPhysicalDevices(pa,size,op,wdata) as
    if (size=256) then
        data0 ← AccessPhysicalDevices(pa,128,op,wdata|27..0)
        data1 ← AccessPhysicalDevices(pa+16,128,op,wdata|255..128)
        data ← data1 || data0
    elseif (0xFFFFFFFF0B000000 ≤ pa ≤ 0xFFFFFFFF0BFFFFFF) then
        //don't perform RMW on this region
        data ← AccessPhysicalOtherBus(pa,size,op,wdata)
    endif
enddef

```

```

elseif (op=W) and (size<128) then
  //this code should change to check pa4_0=0 and size=sizeofreg
  rdata ← AccessPhysicalDevices(pa and -15,128,R,Q)
  bs ← 8*(pa and 15)
  be ← bs + size
  hdata ← rdata[27..be || wdata[be-1..bs || rdata[bs-1..0]
  data ← AccessPhysicalDevices(pa and -15,128,W,hdata)
elseif (0x0000000010000000 ≤ pa ≤ 0xFFFFFFFFFFFFFFFF) then
  data ← 0
elseif (0xFFFFFFFF00000000 ≤ pa ≤ 0xFFFFFFFF08FFFFFF) then
  data ← AccessPhysicalLOC(pa,op,wdata)
elseif (0xFFFFFFFF09000000 ≤ pa ≤ 0xFFFFFFFF09FFFFFF) then
  data ← AccessPhysicalLOCRedundancy(pa,op,wdata)
elseif (0xFFFFFFFF0A000000 ≤ pa ≤ 0xFFFFFFFF0AFFFFFF) then
  data ← AccessPhysicalLTB(pa,op,wdata)
elseif (0xFFFFFFFF0C000000 ≤ pa ≤ 0xFFFFFFFF0CFFFFFF) then
  data ← AccessPhysicalGTB(pa,op,wdata)
elseif (0xFFFFFFFF0D000000 ≤ pa ≤ 0xFFFFFFFF0DFFFFFF) then
  data ← AccessPhysicalGTBRegisters(pa,op,wdata)
elseif (0xFFFFFFFF0E000000 ≤ pa ≤ 0xFFFFFFFF0EFFFFFF) then
  data ← AccessPhysicalEventMask(pa,op,wdata)
elseif (0xFFFFFFFF0F000000 ≤ pa ≤ 0xFFFFFFFF0FFFFFFF) then
  data ← AccessPhysicalSpecialRegisters(pa,op,wdata)
elseif (0xFFFFFFFF10000000 ≤ pa ≤ 0xFFFFFFFFFFFFFFFF) then
  data ← 0
elseif (0xFFFFFFFFF0000000 ≤ pa ≤ 0xFFFFFFFFFFFFFFFF) then
  data ← AccessPhysicalROM(pa,op,wdata)
endif
enddef

def data ← AccessPhysicalSpecialRegisters(pa,op,wdata) as
  if (pa7_0 ≥ 0x10) then
    data ← 0
  elseif (0xFFFFFFFFF0000000 ≤ pa ≤ 0xFFFFFFFFF0003FFF) then
    data ← AccessPhysicalEventRegister(pa,op,wdata)
  elseif (0xFFFFFFFFF0005000 ≤ pa ≤ 0xFFFFFFFFF0005FFF) then
    data ← AccessPhysicalThread(pa,op,wdata)
  elseif (0xFFFFFFFFF0004000 ≤ pa ≤ 0xFFFFFFFFF0007FFF) then
    data ← AccessPhysicalClock(pa,op,wdata)
  elseif (0xFFFFFFFFF0008000 ≤ pa ≤ 0xFFFFFFFFF000BFFF) then
    data ← AccessPhysicalTally(pa,op,wdata)
  elseif (0xFFFFFFFFF000C000 ≤ pa ≤ 0xFFFFFFFFF000CFFF) then
    data ← AccessPhysicalExceptionBase(pa,op,wdata)
  elseif (0xFFFFFFFFF000D000 ≤ pa ≤ 0xFFFFFFFFF000DFFF) then
    data ← AccessPhysicalBusControl(pa,op,wdata)
  elseif (0xFFFFFFFFF000E000 ≤ pa ≤ 0xFFFFFFFFF000EFFF) then
    data ← AccessPhysicalStatus(pa,op,wdata)
  elseif (0xFFFFFFFFF000F000 ≤ pa ≤ 0xFFFFFFFFF000FFFF) then
    data ← AccessPhysicalControl(pa,op,wdata)
  endif
enddef

```

Architecture Description Register

The last hexlet of the internal ROM contains data that describes implementation-dependent choices within the architecture specification. The last quadlet of the internal ROM contains a branch-immediate instruction, so the architecture description is limited to 96 bits.

Address range	bytes	Meaning
FFFF FFFF FFFF FFFC_FFFF FFFF FFFF FFFF	4	Reset address
FFFF FFFF FFFF FFF0_FFFF FFFF FFFF FFFF	12	Architecture Description Register

The table below indicates the detailed layout of the Architecture Description Register.

bits	field name	value range		interpretation
127..96	bi start			Contains a branch instruction for bootstrap from internal ROM
95..23	0	0	0	reserved
22..21	GT	1	0..3	\log_2 threads which share a global TB
20..17	GE	7	0..15	\log_2 entries in global TB
16	LB	1	0..1	local TB based on base register
15..14	LE	1	0..3	\log_2 entries in local TB (per thread)
13	CT	1	0..1	dedicated tags in first-level cache
12..10	CS	2	0..7	\log_2 cache blocks in first-level cache set
9..5	CE	9	0..31	\log_2 cache blocks in first-level cache
4..0	T	4	1..31	number of execution threads

The architecture description register contains a machine-readable version of the architecture framework parameters: T, CE, CS, CT, LE, GE, and GT described in the Architectural Framework section on page 17.

Status Register

The status register is a 64-bit register with both read and write access, though the only legal value which may be written is a zero, to clear the register. The result of writing a non-zero value is not specified.

bits	field name	value range		interpretation
63	power-on	1	0..1	This bit is set when a power-on reset has caused a reset.
62	internal reset	0	0..1	This bit is set when writing to the control register caused a reset.
61	bus reset	0	0..1	This bit is set when a bus reset has caused a reset.
60	double check	0	0..1	This bit is set when a double machine check has caused a reset.
59	meltdown	0	0..1	This bit is set when the meltdown detector has caused a reset.
58..56	0	0*	0	Reserved for other machine check causes.
55	event exception	0	0..1	This bit is set when an exception in event thread has caused a machine check.
54	watchdog timeout	0	0..1	This bit is set when a watchdog timeout has caused a machine check.
53	bus error	0	0..1	This bit is set when a bus error has caused a machine check.
52	cache error	0	0..1	This bit is set when a cache error has caused a machine check.
51	vm error	0	0..1	This bit is set when a virtual memory error has caused a machine check.
50..48	0	0*	0	Reserved for other machine check causes.
47..32	machine check detail	0*	0..40 95	Set to exception code if Exception in event thread. Set to bus error code is bus error.
31..0	machine check program counter	0	0	Set to indicate bits 31..0 of the value of the thread 0 program counter at the initiation of a machine check.

The power-on bit of the status register is set upon the completion of a power-on reset.

The bus reset bit of the status register is set upon the completion of a bus reset initiated by the RESET pin of the Socket 7 interface.

The double check bit of the status register is set when a second machine check occurs that prevents recovery from the first machine check, or which is indicative of machine check recovery software failure. Specifically, the occurrence of an event exception, watchdog timeout, bus error, or meltdown while any reset or machine check cause bit of the status register is still set results in a double check reset.

The meltdown bit of the status register is set when the meltdown detector has discovered an on-chip temperature above the threshold set by the meltdown threshold field of the control register, which causes a reset to occur.

The event exception bit of the status register is set when an event thread suffers an exception, which causes a machine check. The exception code is loaded into the machine

check detail field of the status register, and the machine check program counter is loaded with the low-order 32 bits of the program counter and privilege level.

The watchdog timeout bit of the status register is set when the watchdog timer register is equal to the clock cycle register, causing a machine check.

The bus error bit of the status register is set when a bus transaction error (bus timeout, invalid transaction code, invalid address, parity errors) has caused a machine check.

The cache error bit of the status register is set when a cache error, such as a cache parity error has caused a machine check.

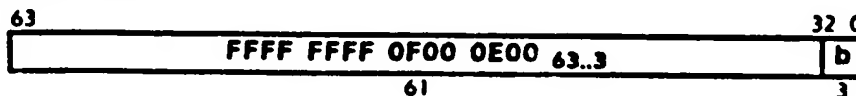
The vm error bit of the status register is set when a virtual memory error, such as a GTB multiple-entry selection error has caused a machine check.

The machine check detail field of the status register is set when a machine check has been completed. For an exception in event thread, the value indicates the type of exception for which the most recent machine check has been reported. For a bus error, this field may indicate additional detail on the cause of the bus error. For a cache error, this field may indicate the address of the error at which the cache parity error was detected.

The machine check program counter field of the status register is loaded with bits 31..0 of the program counter and privilege level at which the most recent machine check has occurred. The value in this field provides a limited diagnostic capability for purposes of software development, or possibly for error recovery.

Physical address

The physical address of the Status Register, byte b is:



Definition

```
def data ← AccessPhysicalStatus(pa,op,wdata) as
  case op of
    R:
      data ← 064 || StatusRegister
    W:
      StatusRegister ← wdata63..0
  endcase
enddef
```

Control Register

The control register is a 64-bit register with both read and write access. It is altered only by write access to this register.

bits	field name	value	range	interpretation
63	reset	0	0..1	set to invoke internal reset
62	MMU	0	0..1	set to enable the MMU
61	LOC parity	0	0..1	set to enable LOC parity
60	meltdown	0	0..1	set to enable meltdown detector
59..57	LOC timing	0	0..7	adjust LOC timing 0⇒slow..7⇒fast
56..55	LOC stress	0	0..3	adjust LOC stress 0⇒normal
54..52	clock timing	0	0..7	adjust clock timing 0⇒slow..7⇒fast
51..12	0	0	0	Reserved
11..8	global access	0*	0..15	global access
7..0	niche limit	0*	0..12	niche limit
			7	

The reset bit of the control register provides the ability to reset an individual Zeus device in a system. Writing a one (1) to this bit is equivalent to a power-on reset or a bus reset. The duration of the reset is sufficient for the operating state changes to have taken effect. At the completion of the reset operation, the internal reset bit of the status register is set and the reset bit of the control register is cleared (0).

The MMU bit of the control register provides the ability to enable or disable the MMU features of the Zeus processor. Writing a zero (0) to this bit disables the MMU, causing all MMU-related exceptions to be disabled and causing all load, store, program and gateway virtual addresses to be treated as physical addresses. Writing a one (1) to this bit enables the MMU and MMU-related exceptions. On a reset or machine check, this bit is cleared (0), thus disabling the MMU.

The parity bit of the control register provides the ability to enable or disable the cache parity feature of the Zeus processor. Writing a zero (0) to this bit disables the parity check, causing the parity check machine check to be disabled. Writing a one (1) to this bit enables the cache parity machine check. On a reset or machine check, this bit is cleared (0), thus disabling the cache parity check.

The meltdown bit of the control register provides the ability to enable or disable the meltdown detection feature of the Zeus processor. Writing a zero (0) to this bit disables the meltdown detector, causing the meltdown detected machine check to be disabled. Writing a one (1) to this bit enables the meltdown detector. On a reset or machine check, this bit is cleared (0), thus disabling the meltdown detector.

The LOC timing bits of the control register provide the ability to adjust the cache timing of the Zeus processor. Writing a zero (0) to this field sets the cache timing to its slowest state, enhancing reliability but limiting clock rate. Writing a seven (7) to this field sets the cache timing to its fastest state, limiting reliability but enhancing performance. On a reset or machine check, this field is cleared (0), thus providing operation at low clock rate. Changing this register should be performed when the cache is not actively being operated.

The LOC stress bits of the control register provide the ability to stress the LOC parameters by adjusting voltage levels within the LOC. Writing a zero (0) to this field sets the cache parameters to its normal state, enhancing reliability. Writing a non-zero value (1, 2, or 3) to this field sets the cache parameters to levels at which cache reliability is slightly compromised. The stressed parameters are used to cause LOC cells with marginal performance to fail during self-test, so that redundancy can be employed to enhance reliability. On a reset or machine check, this field is cleared (0), thus providing operation at normal parameters. Changing this register should be performed when the cache is not actively being operated.

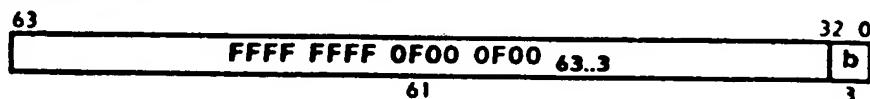
The clock timing bits of the control register provide the ability to adjust the clock timing of the Zeus processor. Writing a zero (0) to this field sets the clock timing to its slowest state, enhancing reliability but limiting clock rate. Writing a seven (7) to this field sets the clock timing to its fastest state, limiting reliability but enhancing performance. On a power on reset, bus reset, or machine check, this field is cleared (0), thus providing operation at low clock rate. The internal clock rate is set to $(\text{clock timing} + 1) / 2 * (\text{external clock rate})$. Changing this register should be performed along with a control register reset.

The global access bits of the control register determine whether a local TB miss cause an exceptions or treatment as a global address. A single bit, selected by the privilege level active for the access from four bit configuration register field, "Global Access," (GA) determines the result. If GAp₁ is zero (0), the failure causes an exception, if it is one (1), the failure causes the address to be used as a global address directly.

The niche limit bits of the control register determine which cache lines are used for cache access, and which lines are used for niche access. For addresses $pa_{14..8} < nl$, a 7-bit address modifier register *am* is inclusive-or'ed against $pa_{14..8}$ to determine the cache line. The cache modifier *am* must be set to $(17 - \log(128 - nl)) \parallel (\log(128 - nl))$ for proper operation. The *am* value does not appear in a register and is generated from the *nl* value.

Physical address

The physical address of the Control Register, byte *b* is:



Definition

```
def data ← AccessPhysicalControl(pa, op, wdata) as
  case op of
    R:
      data ← 064 || ControlRegister
    W:
      ControlRegister ← wdata63..0
  endcase
enddef
```

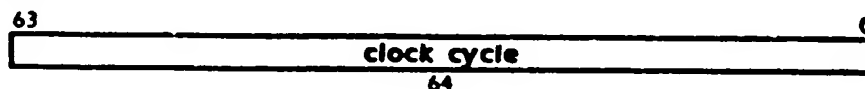
Clock

The Zeus processor provides internal clock facilities using three registers, a clock cycle register that increments one every cycle, a clock event register that sets the clock bit in the event register, and a clock watchdog register that invokes a clock watchdog machine check. These registers are memory mapped.

Clock Cycle

Each Zeus processor includes a clock that maintains processor-clock-cycle accuracy. The value of the clock cycle register is incremented on every cycle, regardless of the number of instructions executed on that cycle. The clock cycle register is 64-bits long.

For testing purposes the clock cycle register is both readable and writable, though in normal operation it should be written only at system initialization time; there is no mechanism provided for adjusting the value in the clock cycle counter without the possibility of losing cycles.

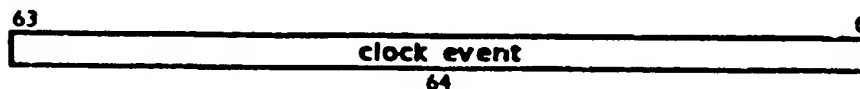


Clock Event

An event is asserted when the value in the clock cycle register is equal to the value in the clock event register, which sets the clock bit in the event register.

It is required that a sufficient number of bits be implemented in the clock event register so that the comparison with the clock cycle register overflows no more frequently than once per second. 32 bits is sufficient for a 4 GHz clock. The remaining unimplemented bits must be zero whenever read, and ignored on write. Equality is checked only against bits that are implemented in both the clock cycle and clock event registers.

For testing purposes the clock event register is both readable and writable, though in normal operation it is normally written to.



Clock Watchdog

A Machine Check is asserted when the value in the clock cycle register is equal to the value in the clock watchdog register, which sets the watchdog timeout bit in the control register.

A Machine Check or a Reset, of any cause including a clock watchdog, disables the clock watchdog machine check. A write to the clock watchdog register enables the clock watchdog machine check.

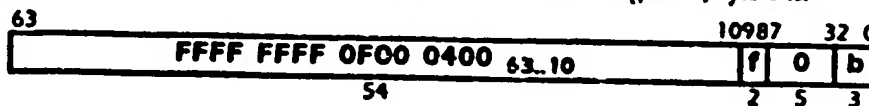
It is required that a sufficient number of bits be implemented in the clock watchdog register so that the comparison with the clock cycle register overflows no more frequently than once per second. 32 bits is sufficient for a 4 GHz clock. The remaining unimplemented bits must be zero whenever read, and ignored on write. Equality is checked only against bits that are implemented in both the clock cycle and clock watchdog registers.

The clock watchdog register is both readable and writable, though in normal operation it is usually and periodically written with a sufficiently large value that the register does not equal the value in the clock cycle register before the next time it is written.



Physical address

The Clock registers appear at three different locations, for which three registers of the Clock are mapped. The Clock Cycle counter is register 0, the Clock Event is register 2, and Clock Watchdog is register 3. The physical address of a Clock Register *f*, byte *b* is:



Definition

```

def data ← AccessPhysicalClock(pa, op, wdata) as
  f ← pa9..8
  case f || op of
    0 || R:
      data ← 064 || ClockCycle
    0 || W:
      ClockCycle ← wdata63..0
    2 || R:
      data ← 096 || ClockEvent
    2 || W:
      ClockEvent ← wdata31..0
    3 || R:
      data ← 096 || ClockWatchdog
    3 || W:
      ClockWatchdog ← wdata31..0
      EnableWatchdog ← 1
  endcase
enddef

def RunClock as
  forever
    ClockCycle ← ClockCycle + 1
    if EnableWatchdog and (ClockCycle31..0 = ClockWatchdog31..0) then
      raise ClockWatchdogMachineCheck
    elsif (ClockCycle31..0 = ClockEvent31..0) then
      EventRegister0 ← 1
  end
end

```

```

    endl
    wait
    endforever
enddel

```

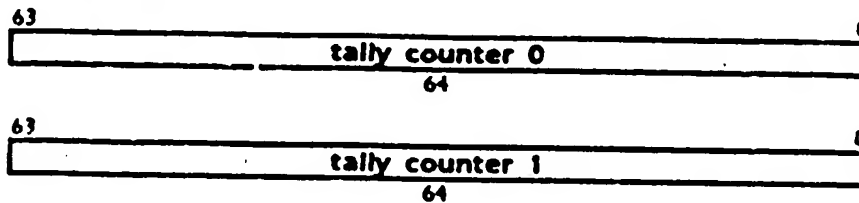
Tally

Tally Counter

Each processor includes two counters that can tally processor-related events or operations. The values of the tally counter registers are incremented on each processor clock cycle in which specified events or operations occur. The tally counter registers do not signal events.

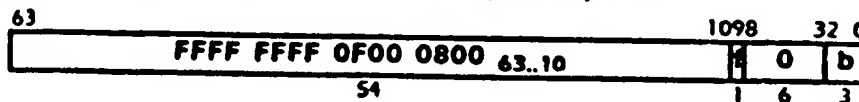
It is required that a sufficient number of bits be implemented so that the tally counter registers overflow no more frequently than once per second. 32 bits is sufficient for a 4 GHz clock. The remaining unimplemented bits must be zero whenever read, and ignored on write.

For testing purposes each of the tally counter registers are both readable and writable, though in normal operation each should be written only at system initialization time; there is no mechanism provided for adjusting the value in the event counter registers without the possibility of losing counts.



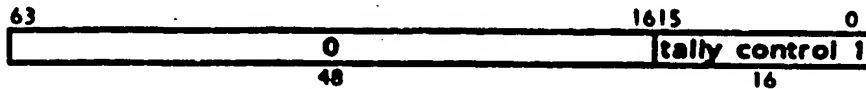
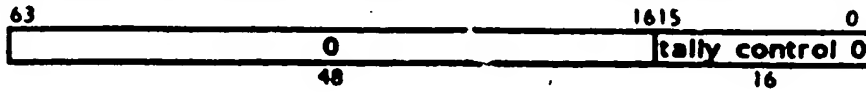
Physical address

The Tally Counter registers appear at two different locations, for which the two registers are mapped. The physical address of a Tally Counter register *f*, byte *b* is:

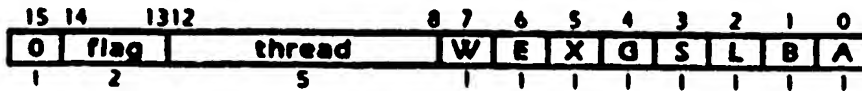


Tally Control

The tally counter control registers each select one metric for one of the tally counters.

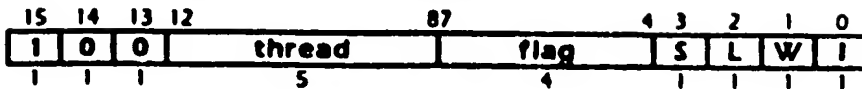


Each control register is loaded with a value in one of the following formats:



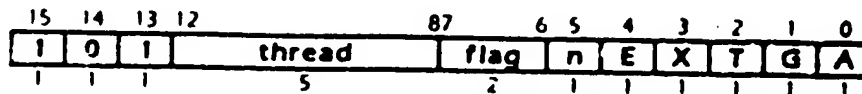
flag	meaning
0	count instructions issued
1	count instructions retired (differs by branch mispred, exceptions)
2	count cycles in which at least one instruction is issued
3	count cycles in which next instruction is waiting for issue

W E X G S L B A: include instructions of these classes



flag	meaning
0	count bytes transferred cache/buffer to/from processor
1	count bytes transferred memory to/from cache/buffer
2	
3	
4	count cache hits
5	count cycles in which at least one cache hit occurs
6	count cache misses
7	count cycles in which at least one cache miss occurs
8..15	

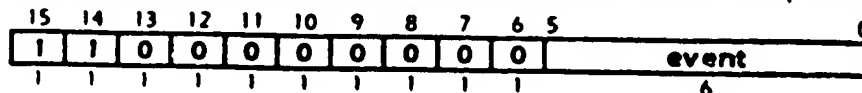
S L W I: include instructions of these classes (Store, Load, Wide, Instruction fetch)



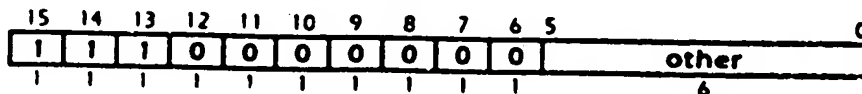
flag	meaning
0	count cycles in which a new instruction is issued
1	count cycles in which an execution unit is busy
2	
3	count cycles in which an instruction is waiting for issue

n select unit number for G or A unit

E X T G A: include units of these classes (Ensemble, Crossbar, Translate, Group, Address)



event: select event number from event register



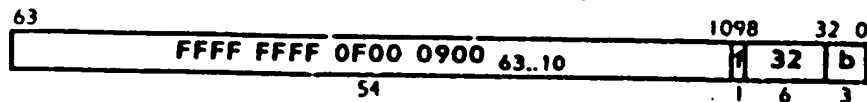
Other valid values for the tally control fields are given by the following table:

other	meaning
0	count number of instructions waiting to issue each cycle
1	count number of instructions waiting in spring each cycle
2..63	Reserved

tally control field interpretation

Physical address

The Tally Control registers appear at two different locations, for which the two registers are mapped. The physical address of a Tally Control register f, byte b is:



Definition

```

def data ← AccessPhysicalTally(pa,op,wdata) as
  f ← pa9
  case pa8 || op of
    0 || R:
      data ← 0% || TallyCounter[f]
    0 || W:
      TallyCounter[f] ← wdata31..0

```

```

1 || R:
    data ← 0112 || TallyControl[]
1 || W:
    TallyControl[] ← wdata15 0
endcase
enddef

```

Thread Register

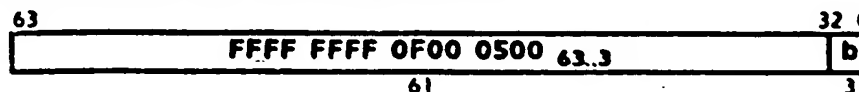
The Zeus processor includes a register that effectively contains the current thread number that reads the register. In this way, threads running identical code can discover their own identity.

It is required that a sufficient number of bits be implemented so that each thread receives a distinct value. Values must be consecutive, unsigned and include a zero value. The remaining unimplemented bits must be zero whenever read. Writes to this register are ignored.



Physical address

The physical address of the Thread Register, byte b is:



Definition

```

def data ← AccessPhysicalThread(pa,op,wdata) as
  case op of
    R:
        data ← 064 || Thread
    W:
        // nothing
  endcase
enddef

```

Index

A

Access detail required

- by global TH96, 109, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315
- by local TH96, 109, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315
- by tag96, 109, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315

Access disallowed

- by global TH96, 108, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315
- by local TH96, 108, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315
- by tag96, 108, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315
- by virtual address94, 96, 108, 111, 116, 119, 122, 125, 127, 130, 132, 134, 287, 293, 302, 307, 310, 312, 315

Address	73
add	
immediate	80
signed	
check overflow	80
unsigned	
check overflow	80
signed	
check overflow	73
unsigned	
check overflow	73
add	73
and	73
immediate	80
not	73
and not	
immediate	80
compare	
and	
equal zero	76
not equal zero	76
equal	76
zero	76
greater	
signed	76
unsigned	76
zero signed	76
greater equal	
signed	76
unsigned	76
zero signed	76
less	
signed	76
unsigned	76

zero signed	76
less equal	
signed	76
unsigned	76
zero signed	76
not equal	76
zero	76
Compare	76
copy	80
copy immediate	79
Copy Immediate	79
exclusive not	73
fixed point arithmetic exception	76
Immediate	80
Reversed	83
multiples	93
negate	83
signed check overflow	83
no operation	76
not	80
not and	73
immediate	80
not or	73
immediate	80
or 73	
immediate	80
not	73
or not	
immediate	80
Reversed	86
set	79
and	
equal zero	86
not equal zero	86
equal	86
immediate	83
equal zero	86
greater	
immediate	
unsigned	83
signed	86
unsigned	86
zero signed	86
greater equal	
immediate	
signed	83
unsigned	83
signed	86
unsigned	86
zero signed	86
less	
immediate	
signed	83
unsigned	83

signed.....	86
unsigned.....	86
zero signed.....	86
less equal	
immediate.....	83
signed.....	83
unsigned.....	86
zero signed.....	86
not equal.....	86
immediate.....	83
not equal zero.....	86
set and	
equal	
immediate.....	83
not equal	
immediate.....	83
Shift	
Left	
Immediate	
Add.....	89
Subtract.....	90
Shift Immediate.....	91
shift left	
immediate.....	91
add.....	89
signed check overflow.....	91
subtract.....	90
unsigned check overflow.....	91
shift right	
immediate	
unsigned	
signed	
shift right	
immediate.....	91
subtract.....	86
signed check overflow.....	86
unsigned check overflow.....	86
subtract immediate.....	83
signed	
check overflow.....	83
unsigned	
check overflow.....	83
Ternary.....	93
xor	
immediate.....	80
xor.....	73
immediate.....	80
zero.....	79
Always reserved.....	72
Always Reserved.....	72
architecture / description registers.....	407
Arithmetic Operations.....	39

B

Branch.....	21, 94
and	
equal zero.....	98
not equal zero.....	98
back.....	95
barrier.....	97
Conditional.....	98
Floating-Point.....	101
Visibility	
Floating-Point.....	103
down.....	105
equal.....	98
floating point	
double.....	101
half.....	101
quad.....	101
single.....	101
equal zero.....	98
gateway.....	106
Gateway.....	106
greater	
floating point	
double.....	101
half.....	101
quad.....	101
single.....	101
signed.....	98
unsigned.....	98
zero	
signed.....	98
greater equal	
floating point	
double.....	101
half.....	101
quad.....	101
single.....	101
signed.....	98
unsigned.....	98
zero signed.....	98
halt.....	110
limit.....	111
Immediate.....	112
immediate.....	113
link.....	114
Immediate.....	113
link.....	114
invisible	
floating point	
single.....	103
less	
floating point	
double.....	101
half.....	101
quad.....	101
single.....	101

unsigned	98
zero	
signed	98
less equal	
floating-point	
double	101
half	101
quad	101
single	101
signed	98
unsigned	98
zero signed	98
less greater	
floating-point	
double	101
half	101
quad	101
single	101
link	115
lank	115
no operation	98
not equal	98
not equal zero	98
not visible	
floating-point	
single	103
not visible	
floating-point	
single	103
signed	
less	98
visible	
floating-point	
single	103
Branch back	95
Branch barrier	97
Branch Conditionally	23
Branch down	105
Branch halt	110

C

checkpoint	401, 402
Classical Pipeline Structures	49
Clock	
Cycle	412
Event	412
Wakeup	412
Compare-set	23
control register	409, 410, 411
Crossbar	187
compress	
immediate	
signed	
bytes	206
doublets	206
hexlet	206

nibbles	206
octets	206
pecks	206
quadrants	206
unsigned	
bytes	206
doublets	206
hexlet	206
nibbles	206
octets	206
pecks	206
quadrants	206
signed	
bytes	187
doublets	187
hexlet	187
nibbles	187
octets	187
pecks	187
quadrants	187
unsigned	
bytes	187
doublets	187
hexlet	187
nibbles	187
octets	187
pecks	187
quadrants	187
copy	207
deposit	
merge	
bits	201
bytes	201
doublets	201
hexlet	201
nibbles	201
octets	201
pecks	201
quadrants	201
signed	
bytes	196
doublets	196
hexlet	196
nibbles	196
octets	196
pecks	196
quadrants	196
expand	
immediate	
signed	
bytes	206
doublets	206
hexlet	206
nibbles	206
octets	206
pecks	206
quadrants	206

unsigned			rotate left	
bytes	216		immediate	
doublets	216		bytes	216
hexlet	216		doublets	216
nibbles	216		hexlet	216
octlets	216		nibbles	216
pecks	216		octlets	216
quadrlets	216		pecks	216
signed			quadrlets	216
bytes	187		rotate right	
doublets	187		immediate	
hexlet	187		bytes	216
nibbles	187		doublets	216
octlets	187		hexlet	217
pecks	187		nibbles	216
quadrlets	187		octlets	217
unsigned			pecks	216
bytes	187		quadrlets	217
doublets	187		select bytes	220
hexlet	187		shift left	
nibbles	187		bytes	188
octlets	187		doublets	188
pecks	187		hexlet	188
quadrlets	187		immediate	
extend			bytes	217
immediate			doublets	217
unsigned			hexlet	217
bytes	197		nibbles	217
doublets	197		octlets	217
hexlet	197		pecks	217
nibbles	197		quadrlets	217
octlets	197		signed bytes	
pecks	197		check overflow	217
quadrlets	197		signed doublets	
extract	192		check overflow	217
Extract	192		signed hexlet	
Field	196		check overflow	217
Field Inplace	201		signed nibbles	
Inplace	214		check overflow	217
no operation	217		signed octlets	
rotate			check overflow	217
left			signed pecks	
bytes	187		check overflow	217
doublets	187		signed quadrlets	
hexlet	187		check overflow	217
nibbles	187		unsigned bytes	
octlets	187		check overflow	217
pecks	187		unsigned doublets	
quadrlets	187		check overflow	217
right			unsigned hexlet	
bytes	187		check overflow	217
doublets	187		unsigned nibbles	
hexlet	188		check overflow	217
nibbles	187		unsigned octlets	
octlets	188		check overflow	217
pecks	187		unsigned pecks	
quadrlets	187		check overflow	217

unsigned quadlets		hexlet	207
check overflow	207	nibbles	207
merge		octlets	207
bytes	204	pecks	207
doublets	204	quadlets	207
hexlet	204	merge	
immediate		bytes	204
bytes	211	doublets	204
doublets	211	hexlet	204
hexlet	211	immediate	
nibbles	211	bytes	211
octlets	211	doublets	211
pecks	211	hexlet	211
quadlets	211	nibbles	211
nibbles	204	octlets	211
octlets	204	pecks	211
pecks	204	quadlets	211
quadlets	204	nibbles	204
nibbles	188	octlets	204
octlets	188	pecks	204
pecks	188	quadlets	204
quadlets	188	unsigned	
signed		bytes	188
bytes		doublets	188
check overflow	188	hexlet	188
doublets		nibbles	188
check overflow	188	octlets	188
hexlet		pecks	188
check overflow	188	quadlets	188
nibbles		Short Immediate	206
check overflow	188	Inplace	211
octlets		shuffle	
check overflow	188	within bytes	213
pecks		within doublets	213
check overflow	188	within hexlet	213
quadlets		within octlets	213
check overflow	188	within pecks	213
unsigned		within quadlets	213
bytes		within triplet	213
check overflow	188	Shuffle	213
doublets		signed	
check overflow	188	shift right	
hexlet		bytes	188
check overflow	188	doublets	188
nibbles		hexlet	188
check overflow	188	immediate	
octlets		bytes	207
check overflow	188	doublets	207
pecks		hexlet	207
check overflow	188	nibbles	207
quadlets		octlets	207
check overflow	188	pecks	207
shift right		quadlets	207
immediate		nibbles	188
unsigned		octlets	188
bytes	207	pecks	188
doublets	207	quadlets	188

rotate	219
rotate	219
ternary	220
unsigned	
deposit	
bytes	196
doublets	196
hexlet	196
nibbles	196
octets	196
pecks	196
quadrlets	196
extend	
immediate	
bytes	197
doublets	197
hexlet	197
nibbles	197
octets	197
pecks	197
quadrlets	197
withdraw	
bytes	196
doublets	196
hexlet	196
nibbles	196
octets	196
pecks	196
quadrlets	196
withdraw	
bytes	196
doublets	196
hexlet	196
nibbles	196
octets	196
pecks	196
quadrlets	196

D

Data-handling Operations	36
--------------------------	----

E

Ensemble	221
absolute value	
floating-point	
double	276
half	276
quad	276
single	276
add	
floating-point	
double	258
half	258
quad	258
single	258

complex	
multiply	
bytes	221
doublets	221
quadrlets	221
convert	
floating-point	
double from octets	277
double from quad	276
double from single	277
doublets from half	278
half from doublets	276, 277
half from single	276
hexlet from quad	278
hexlet from quad	278
octets from double	278
octets from double	278
quad from double	277
quad from hexlet	277
quadrlets from single	278
single from double	276
single from half	277
single from quadrlets	277
floating-point	
single from double	276
convolve	
complex	
bytes	221
doublets	221
floating-point	
double	
big-endian	232
little-endian	232
half	
big-endian	232
little-endian	232
single	
big-endian	232
little-endian	232
quadrlets	221
extract	
immediate	
mixed-signed bytes	
big-endian	
ceiling	226
floor	226
nearest	226
zero	226
little-endian	
ceiling	226
floor	226
nearest	226
zero	227
mixed-signed doublets	
big-endian	
ceiling	226
floor	226

nearest	226
zero	226
little-endian	
ceiling	227
floor	227
nearest	227
zero	227
mixed-signed octets	
big-endian	
ceiling	226
floor	226
nearest	226
zero	226
little-endian	
ceiling	227
floor	227
nearest	227
zero	227
mixed-signed quadrlets	
big-endian	
ceiling	226
floor	226
nearest	226
zero	226
little-endian	
ceiling	227
floor	227
nearest	227
zero	227
signed bytes	
big-endian	
ceiling	225
floor	225
nearest	225
zero	225
little-endian	
ceiling	226
floor	226
nearest	226
zero	226
signed complex bytes	
big-endian	
ceiling	225
floor	225
nearest	225
zero	225
little-endian	
ceiling	225
floor	225
nearest	225
zero	225
signed complex doublets	
big-endian	
ceiling	225
floor	225
nearest	225
zero	225

little-endian	
ceiling	225
floor	225
nearest	225
zero	225
signed complex octets	
big-endian	
ceiling	225
floor	225
nearest	225
zero	225
little-endian	
ceiling	225
floor	225
nearest	225
zero	225
signed complex quadrlets	
big-endian	
ceiling	225
floor	225
nearest	225
zero	225
little-endian	
ceiling	225
floor	225
nearest	225
zero	225
signed doublets	
big-endian	
ceiling	225
floor	225
nearest	226
zero	226
little-endian	
ceiling	226
floor	226
nearest	226
zero	226
signed octets	
big-endian	
ceiling	226
floor	226
nearest	226
zero	226
little-endian	
ceiling	226
floor	226
nearest	226
zero	226
signed quadrlets	
big-endian	
ceiling	226
floor	226
nearest	226
zero	226
little-endian	
ceiling	226

floor	226	quadlets	221
nearest	226	unsigned	
zero	226	bytes	221
unsigned bytes		doublets	221
big-endian		octlets	221
ceiling	227	quadlets	221
floor	227	Convolve	
nearest	227	Extract	
little-endian		Immediate	225
ceiling	227	Floating-Point	232
floor	227	copy	
nearest	227	floating-point	
unsigned doublets		double	276
big-endian		half	276
ceiling	227	quad	276
floor	227	single	276
nearest	227	divide	
little-endian		floating-point	
ceiling	227	double	258, 259
floor	227	half	258
nearest	227	quad	259
unsigned octlets		single	258
big-endian		signed	
ceiling	227	octlets	221
floor	227	unsigned	
nearest	227	octlets	221
little-endian		l'assembleconvert	
ceiling	227	floating-point	
floor	227	doubletsfromhalf	278
nearest	227	octletsfromdouble	278
unsigned quadlets		quadletsfromsingle	278
big-endian		extract	236
ceiling	227	immediate	
floor	227	signed	
nearest	227	bytes	
little-endian		ceiling	244
ceiling	227	floor	244
floor	227	nearest	244
nearest	227	zero	244
floating-point double		doublets	
big-endian	232	ceiling	244
little-endian	232	floor	244
floating-point half		nearest	244
big-endian	232	zero	244
little-endian	232	octlets	
floating-point single		ceiling	244
big-endian	232	floor	244
little-endian	232	nearest	244
mixed-signed		zero	244
bytes	221	quadlets	
doublets	221	ceiling	244
octlets	221	floor	244
quadlets	221	nearest	244
signed		zero	244
bytes	221	unsigned	
doublets	221	bytes	
octlets	221	ceiling	244

floor	244	unsigned	
nearest	244	bytes	252
doublets		doublets	252
ceiling	244	octlets	252
floor	244	quadlets	252
nearest	244	complex	
octlets		floating-point	
ceiling	244	double	259
floor	244	half	259
nearest	244	single	259
quadlets		extract	256
ceiling	244	immediate	
floor	244	mixed-signed	
nearest	244	bytes	
Extract	256	ceiling	245
immediate	244	floor	245
immediate inplace	251	nearest	245
floating-point		zero	245
reciprocal square root estimate		doublets	
double	278	ceiling	245
half	277	floor	245
quad	278	nearest	245
single	277	zero	245
floating-point	258	octlets	
inplace	261	ceiling	245
floating-point	264	floor	245
log of most significant bit		nearest	245
signed		zero	245
bytes	274	quadlets	
doublets	274	ceiling	245
halflets	274	floor	245
octlets	274	nearest	245
quadlets	274	zero	245
unsigned		signed	
bytes	274	bytes	
doublets	274	ceiling	244
halflets	274	floor	244
octlets	274	nearest	244
quadlets	274	zero	244
multiply		doublets	
add		ceiling	244
extract		floor	244
immediate		nearest	244
mixed-signed		zero	244
bytes	251	octlets	
doublets	251	ceiling	245
octlets	251	floor	245
quadlets	251	nearest	245
signed		zero	245
bytes	251	quadlets	
complex bytes	251	ceiling	244
complex doublets	251	floor	244
complex octlets	251	nearest	245
complex quadlets	251	zero	245
doublets	251, 252	unsigned	
octlets	252	bytes	
quadlets	252	ceiling	245

floor	245	bytes and doublets	261
nearest	245	doublets and quadlets	261
doublets		quadlets and octlets	261
ceiling	245	complex floating-point	
floor	245	double	264
nearest	245	half	264
octlets		single	264
ceiling	246	floating-point	
floor	246	double	264
nearest	246	half	264
quadlets		quad	264
ceiling	245	single	264
floor	245	mixed-signed	
nearest	246	bytes and doublets	261
floating-point		doublets and quadlets	261
double	259	octlets and hexlet	261
half	259	quadlets and octlets	261
quad	259	signed	
single	259	bytes and doublets	261
mixed-signed		doublets and quadlets	261
bytes	221	octlets and hexlet	261
doublets	221	quadlets and octlets	261
octlets	221	unsigned	
quadlets	221	bytes and doublets	261
polynomial		doublets and quadlets	261
bytes	221	octlets and hexlet	261
doublets	221	quadlets and octlets	261
octlets	221	multiply extract	
quadlets	221	immediate	
signed		complex	
bytes	221	bytes	
doublets	221	ceiling	245
octlets	221	floor	245
quadlets	221	nearest	245
sum		zero	245
mixed-signed		doublets	
bytes		ceiling	245
doublets		floor	245
octlets		nearest	245
quadlets		zero	245
signed		octlets	
bytes		ceiling	245
doublets		floor	245
octlets		nearest	245
quadlets		zero	245
unsigned		quadlets	
bytes		ceiling	245
doublets		floor	245
octlets		nearest	245
quadlets		zero	245
multiply add		multiply subtract	
complex		complex	
bytes	222	bytes and doublets	261
doublets	222	doublets and quadlets	261
octlets	222	quadlets and octlets	261
quadlets	222	complex floating-point	
multiply add		double	264
complex			

half	264
single	264
floating-point	
double	264
half	264
quad	264
single	264
mixed-signed	
bytes and doublets	261
doublets and quadlets	261
octets and hexlet	261
quadlets and octets	261
signed	
bytes and doublets	261
doublets and quadlets	261
octets and hexlet	261
quadlets and octets	261
unsigned	
bytes and doublets	261
doublets and quadlets	261
octets and hexlet	261
quadlets and octets	261
multiply sum	
complex	
bytes	
doublets	
quadlets	
negate	
floating-point	
double	277
half	277
quad	277
single	277
reciprocal estimate	
floating-point	
double	277
half	277
quad	277
single	277
Reversed	
floating-point	
scale add extract	236
square root	
floating-point	
double	279
half	278
quad	279
single	279
subtract	
floating-point	
double	267
half	267
quad	267
single	267
sum	
floating-point	
double	279

half	279
quad	279
single	279
signed	
bytes	274
doublets	274
octets	274
quadlets	274
unsigned	
bits	274
bytes	274
doublets	274
octets	274
quadlets	274
Ternary	269
Ternary	
floating-point	272
Unary	274
floating-point	276
Ensemble absolute value	
floating-point	
half	276
Ensemble copy	
floating-point	
half	276
Ensemble scale add	
floating-point	
double	272
half	272
single	272

F

Fixed-point	21
Fixed-point arithmetic	75, 78, 82, 85, 88, 92, 137, 153, 162, 167, 175, 191, 210
Floating-point	22
Floating-point arithmetic	156, 178, 260, 268, 282, 307
Forwarding	55

G

Galois Field Operations	39
Gateway	18
Gateway disallowed	108
Global TB miss	96, 109, 119, 122, 125, 127, 134, 287, 294, 302, 307, 310, 312, 315
Group	
add	
bits	141
bytes	135
doublets	135
half	
signed	
bytes	
ceiling	138

floor.....	138	quadlet	
nearest.....	138	check overflow.....	159
zero.....	138	unsigned	
doublets		doublet	
ceiling.....	138	check overflow.....	159
floor.....	138	hexlet	
nearest.....	138	check overflow.....	159
zero.....	138	octlet	
hexlet		check overflow.....	159
ceiling.....	138	quadlet	
floor.....	138	check overflow.....	159
nearest.....	138	limit	
zero.....	138	signed	
octets		bytes.....	135
ceiling.....	138	doublets.....	135
floor.....	138	hexlet.....	135
nearest.....	138	octlets.....	135
zero.....	138	quadlets.....	135
quadlets		unsigned	
ceiling.....	138	bytes.....	135
floor.....	138	doublets.....	135
nearest.....	138	hexlet.....	135
zero.....	138	octlets.....	135
unsigned		quadlets.....	135
bytes		octets.....	135
ceiling.....	138	quadlets.....	135
floor.....	138	signed	
nearest.....	138	bytes	
doublets		check overflow.....	135
ceiling.....	138	doublets	
floor.....	138	check overflow.....	135
nearest.....	138	hexlet	
hexlet		check overflow.....	135
ceiling.....	138	octets	
floor.....	138	check overflow.....	135
nearest.....	138	quadlets	
octets		check overflow.....	135
ceiling.....	138	unsigned	
floor.....	138	bytes	
nearest.....	138	check overflow.....	135
quadlets		doublets	
ceiling.....	138	check overflow.....	135
floor.....	138	hexlet	
nearest.....	138	check overflow.....	135
hexlet.....	135	octets	
immediate		check overflow.....	135
doublet.....	159	quadlets	
hexlet.....	159	check overflow.....	135
octlet.....	159	Add.....	135
quadlet.....	159	lshv.....	138
signed		add add add	
doublet		bits.....	141
check overflow.....	159	bytes.....	168
hexlet		doublets.....	168
check overflow.....	159	hexlet.....	168
octlet		octets.....	168
check overflow.....	159	quadlets.....	168

add add subtract		octlets	150
bits	141	quadlets	150
bytes	168	greater	
doublets	168	floating-point	
hexlet	168	double	155
octlets	168	half	155
quadlets	168	quad	155
add subtract add		single	155
bytes	168	unsigned	
doublets	168	bytes	150
hexlet	168	doublets	150
octlets	168	hexlet	150
quadlets	168	octlets	150
and	141	quadlets	150
immediate		zero	
doublet	159	signed	
hexlet	159	bytes	150
octlet	159	doublets	150
quadlet	159	hexlet	150
and not	141	octlets	150
immediate		quadlets	150
doublet	160	greater equal	
hexlet	160	signed	
octlet	160	bytes	148
quadlet	160	doublets	148
boolean	141	hexlet	148
Boolean	141	octlets	148
compare		quadlets	148
and		unsigned	
equal zero		bytes	148
bytes	148	doublets	148
doublets	148	hexlet	148
hexlet	148	octlets	148
octlets	148	quadlets	148
quadlets	148	zero signed	
not equal zero		bytes	150
bytes	148	doublets	150
doublets	148	hexlet	150
hexlet	148	octlets	150
octlets	148	quadlets	150
quadlets	148	greater or equal	
equal		floating-point	
bytes	148	double	154
doublets	148	half	154
floating-point		quad	154
double	154	single	154
half	154	less	
quad	154	floating-point	
single	154	double	154
hexlet	148	half	154
octlets	148	quad	154
quadlets	148	single	154
zero		unsigned	
signed		bytes	148
bytes	150	doublets	148
doublets	150	hexlet	148
hexlet	150	octlets	148

quadlets	148	less	
zero		bytes	148
signed		doublets	148
bytes	150	hexlet	148
doublets	150	octlets	148
hexlet	150	quadlets	148
octlets	150	Compare	148
quadlets	150	floating-point	
less equal		copy	141, 160
floating-point		copy immediate	
double	155	byte	157
half	155	doublet	157
quad	155	hexlet	157
single	155	octlet	157
signed		quadlet	157
bytes	150	Copy Immediate	157
doublets	150	exclusive-nor	142
hexlet	150	immediate	
octlets	150	doublet	160
quadlets	150	hexlet	160
unsigned		octlet	160
bytes	150	quadlet	160
doublets	150	exclusive-or	142
hexlet	150	immediate	
octlets	150	doublet	159
quadlets	150	hexlet	159
zero signed		octlet	159
bytes	150	quadlet	159
doublets	150	fixed point arithmetic exception	151
hexlet	151	Galois field multipl'y	
octlets	150	byte	269
quadlets	150	octlet	269
less or greater		Immediate	159
floating-point		Reversed	163
double	154	Inplace	168
half	154	multiplex	186
quad	154	nand	141
single	154	negate	
not equal		doublet	164
bytes	148	hexlet	164
doublets	148	octlet	164
hexlet	149	quadlet	164
octlets	148	signed	
quadlets	148	doublet	
zero signed		check overflow	164
bytes	151	hexlet	
doublets	151	check overflow	164
hexlet	151	octlet	
octlets	151	check overflow	164
quadlets	151	quadlet	
signed		check overflow	164
greater		no operation	151
bytes	150	nor	141
doublets	150	not	141, 160
hexlet	150	not and	
octlets	150	immediate	
quadlets	150	doublet	159

hexlet.....	159
octlet.....	159
quadlet.....	159
not or	
immediate	
doublet.....	159
hexlet.....	159
octlet.....	159
quadlet.....	159
or 141	
immediate	
doublet.....	159
hexlet.....	159
octlet.....	159
quadlet.....	159
or not.....	141
immediate	
doublet.....	160
hexlet.....	160
octlet.....	160
quadlet.....	160
Reversed.....	170
Floating-point	
set.....	157
and	
equal zero	
bits.....	141
bytes.....	170
doublets.....	170
hexlet.....	170
octlets.....	170
quadlets.....	170
not equal zero	
bits.....	141
bytes.....	170
doublets.....	170
hexlet.....	170
octlets.....	170
quadlets.....	170
equal	
bits.....	141
bytes.....	170
doublets.....	170
floating-point	
double.....	176
half.....	176
quad.....	176
single.....	176
hexlet.....	170
octlets.....	170
quadlets.....	170
zero	
bytes.....	172
doublets.....	172
hexlet.....	172
octlets.....	172
quadlets.....	172

greater	
floating-point	
double.....	177
half.....	177
quad.....	177
single.....	177
immediate	
unsigned	
doublets.....	164
hexlet.....	164
octlets.....	164
quadlets.....	164
signed	
bits.....	141
unsigned	
bits.....	141
bytes.....	173
doublets.....	173
hexlet.....	173
octlets.....	173
quadlets.....	173
zero	
signed	
bits.....	141
bytes.....	172
doublets.....	172
hexlet.....	172
octlets.....	172
quadlets.....	172
greater equal	
floating-point	
double.....	176
half.....	176
quad.....	176
single.....	176
signed	
bits.....	141
bytes.....	170
doublets.....	170
hexlet.....	170
octlets.....	170
quadlets.....	170
unsigned	
bits.....	141
bytes.....	170
doublets.....	170
hexlet.....	170
octlets.....	170
quadlets.....	170
zero signed	
bits.....	141
bytes.....	172
doublets.....	172
hexlet.....	172
octlets.....	172
quadlets.....	172
immediate	

signed		quadlets	172
greater		unsigned	
doublets	164	bits	141
hexlet	164	bytes	172
octlets	164	doublets	172
quadlets	164	hexlet	172
less		octlets	172
floating-point		quadlets	172
double	176	zero signed	
half	176	bits	141
quad	176	hexlet	172
single	176	less greater	
signed		floating-point	
bits	141	double	176
unsigned		half	176
bits	141	quad	176
bytes	170	single	176
doublets	170	not equal	
hexlet	170	bits	141
octlets	170	bytes	170
quadlets	170	doublets	170
zero		hexlet	171
signed		octlets	170
bits	141	quadlets	170
bytes	172	zero	
doublets	172	bytes	172
hexlet	172	doublets	172
octlets	172	hexlet	172
quadlets	172	octlets	172
less equal		quadlets	172
floating-point		signed	
double	177	greater	
half	177	bytes	172
quad	177	doublets	172
single	177	hexlet	173
immediate		octlets	172
signed		quadlets	172
doublets	164	less	
hexlet	164	bytes	170
octlets	164	doublets	170
quadlets	164	hexlet	170
unsigned		octlets	170
doublets	164	quadlets	170
hexlet	164	set	141
octlets	164	or and	
quadlets	164	equal zero	
signed		immediate	
bits	141	doublets	163
bytes	172	hexlet	163
doublets	172	octlets	163
hexlet	172	quadlets	163
octlets	172	not equal zero	
quadlets	172	immediate	
signed zero		doublets	163
bytes	172	hexlet	163
doublets	172	octlets	163
octlets	172	quadlets	163

set equal	
immediate	
doublets	163
hexlet	163
octlets	163
quadlets	163
set greater equal	
immediate	
signed	
doublets	163
hexlet	163
octlets	163
quadlets	163
unsigned	
doublets	163
hexlet	163
octlets	163
quadlets	163
set less	
immediate	
signed	
doublets	163
hexlet	163
octlets	163
quadlets	163
set not equal	
immediate	
doublets	163
hexlet	163
octlets	163
quadlets	163
set signed	
less	
immediate	
doublets	163
hexlet	163
octlets	163
quadlets	163
shift left	
immediate	
add	
bytes	179
doublets	179
hexlet	179
octlets	179
quadlets	179
subtract	
bytes	181
doublets	181
hexlet	181
octlets	181
quadlets	181
Shift Left	
Immediate	
Add	179
Subtract	181
subtract	

bits	142
bytes	171
doublets	171
half	
signed	
bytes	
ceiling	183
floor	183
nearest	183
zero	183
doublets	
ceiling	183
floor	183
nearest	183
zero	183
hexlet	
ceiling	183
floor	183
nearest	183
zero	183
octlets	
ceiling	183
floor	183
nearest	183
zero	183
quadlets	
ceiling	183
floor	183
nearest	183
zero	183
unsigned	
bytes	
ceiling	183
floor	183
nearest	183
zero	183
doublets	
ceiling	183
floor	183
nearest	183
zero	183
hexlet	
ceiling	183
floor	183
nearest	183
zero	184
octlets	
ceiling	183
floor	183
nearest	183
zero	183
quadlets	
ceiling	183
floor	183
nearest	183
zero	183
hexlet	171

brut		
signed		
doublet	171	
hexlet	171	
octlet	171	
quadlet	171	
brut signed		
bytes	171	
octlets	171	
quadlets	171	
signed		
bytes		
check overflow	171	
doublet		
check overflow	171	
hexlet		
check overflow	171	
octlet		
check overflow	171	
quadlet		
check overflow	171	
unsigned		
bytes		
check overflow	171	
doublet		
check overflow	171	
hexlet		
check overflow	171	
octlet		
check overflow	171	
quadlet		
check overflow	171	
Subtract		
Half	183	
subtract add add		
bits	141	
subtract add subtract		
bits	141	
subtract immediate		
doublet	163	
hexlet	163	
octlet	163	
quadlet	163	
signed		
doublet		
check overflow	163	
hexlet		
check overflow	163	
octlet		
check overflow	163	
quadlet		
check overflow	163	
unsigned		
doublet		
check overflow	163	
hexlet		
check overflow	163	
octlet		
check overflow	164	
quadlet		
check overflow	164	
signed		
check overflow	164	
doublet		
check overflow	164	
hexlet		
check overflow	164	
octlet		
check overflow	164	
quadlet		
check overflow	164	
subtract limit		
unsigned		
bytes	171	
doublet	171	
hexlet	171	
octlet	171	
quadlet	171	
subtract subtract add		
bits	142	
subtract subtract subtract		
bits	142	
Ternary	186	
three way		
and		
bits	141	
exclusive-nor	141	
exclusive-or	142	
nand	141	
nor	141	
or 141		
zero	142, 157	

I

implementation defined parameters 17

L

least privileged level 316

Load 117

hexlet

aligned

big endian 117

little endian 117

big endian 117

little endian 117

immediate

hexlet

aligned

big endian 120

little endian 120

big endian 120

little endian 120

signed

byte 120

doublet

aligned

big endian 120

little endian 120

big endian 120

little endian 120

octlet

aligned

big endian.....	120
little endian.....	120
big endian.....	120
little endian.....	120
quadlet	
aligned	
big endian.....	120
little endian.....	120
big endian.....	120
little endian.....	120
unsigned	
byte.....	120
doublet	
aligned	
big endian.....	120
little endian.....	120
big endian.....	120
little endian.....	120
octlet	
aligned	
big endian.....	120
little endian.....	120
big endian.....	120
little endian.....	120
quadlet	
aligned	
big endian.....	120
little endian.....	120
big endian.....	120
little endian.....	120
Immediate.....	120
signed	
byte.....	117
doublet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
octlet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
quadlet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
unsigned	
byte.....	117
doublet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
octlet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
quadlet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
unassigned	
byte.....	117
doublet	
aligned	
big endian.....	117
little endian.....	117

big endian.....	117
little endian.....	117
octlet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
quadlet	
aligned	
big endian.....	117
little endian.....	117
big endian.....	117
little endian.....	117
Load and Store.....	21
Local TH mask 96, 109, 119, 122, 125, 127, 130,	
132, 134, 287, 294, 302, 307, 310, 312, 315	

M

machine check.....	410
Memory Management.....	316
most-privileged level.....	316
multiprocessor.....	316

P

Pipeline Organization.....	49
Prediction.....	54
Procedure Calling Conventions.....	40

R

Register Usage.....	40
Reserved instruction.....	200, 203
Reserved Instruction 94, 96, 97, 105, 102, 110, 116,	
158, 210, 212, 218	
reset.....	399
Resources.....	55
Rounding.....	24

S

set on compare.....	35, 54
Software Conventions.....	40
start vector address.....	399, 400, 402
status register.....	407, 408
Store.....	123
add swap immediate	
octlet	
aligned big-endian.....	131
aligned little-endian.....	131
add swap octlet	
aligned big-endian.....	133
aligned little-endian.....	133
byte.....	123
compare swap immediate	

octlet	
aligned big-endian	131
aligned little-endian	
immediate	131
compare swap octlet	
aligned big-endian	133
aligned little-endian	133
double	
aligned	
big-endian	123
little-endian	123
big-endian	123
little-endian	123
double compare swap octlet	
aligned big-endian	126
aligned little-endian	126
hexlet	
aligned	
big-endian	123
little-endian	123
big-endian	123
little-endian	123
immediate	
byte	128
double	
aligned	
big-endian	128
little-endian	128
big-endian	128
little-endian	128
hexlet	
aligned	
big-endian	128
little-endian	128
big-endian	128
little-endian	128
octlet	
aligned	
big-endian	128
little-endian	128
big-endian	128
little-endian	128
quadlet	
aligned	
big-endian	128
little-endian	128
big-endian	128
little-endian	128
immediate	128
inplace	131
inplace	133
multiplex immediate	
octlet	
aligned big-endian	128
aligned little-endian	128
multiplex octlet	
aligned big-endian	123

aligned little-endian	123
multiplex swap immediate	
octlet	
aligned big-endian	131
aligned little-endian	131
multiplex swap octlet	
aligned big-endian	133
aligned little-endian	133
octlet	
aligned	
big-endian	123
little-endian	123
big-endian	123
little-endian	123
quadlet	
aligned	
big-endian	123
little-endian	123
big-endian	123
little-endian	123
Store Double Compare Swap	126
Superspring Pipeline	51
Superstring Pipeline	50
Superspring Pipeline	52

W

Wide

multiply	
matrix	
extract	
big-endian	288
immediate	
complex	
bytes	295
doublets	296
octlets	296
quadlets	296
signed	
bytes	295
doublets	295
octlets	295
quadlets	295
unsigned	
bytes	297
doublets	297
octlets	297
quadlets	297
little-endian	288
Galois big-endian	308
Galois little-endian	308
mixed signed	
byte	
big-endian	283
little-endian	283
doublet	
big-endian	283

little-endian	283	extract immediate mixed-signed	
quadlet		bytes	
big-endian	283	big-endian	
little-endian	283	ceiling	296
polynomial byte		floor	296
big-endian	283	nearest	296
little-endian	283	zero	296
polynomial doublet		little-endian	
big-endian	283	ceiling	296
little-endian	283	floor	296
polynomial quadlet		nearest	296
big-endian	283	zero	296
little-endian	283	doublets	
signed		big-endian	
byte		ceiling	296
big-endian	283	floor	296
little-endian	283	nearest	296
complex byte		zero	296
big-endian	283	little-endian	
little-endian	283	ceiling	296
complex doublet		floor	296
big-endian	283	nearest	296
little-endian	283	zero	296
doublet		octets	
big-endian	283	big-endian	
little-endian	283	ceiling	296
quadlet		floor	297
big-endian	283	nearest	297
little-endian	283	zero	297
unsigned		little-endian	
byte		ceiling	296
big-endian	283	floor	297
little-endian	283	nearest	297
doublet		zero	297
big-endian	283	quadlets	
little-endian	283	big-endian	
quadlet		ceiling	296
big-endian	283	floor	296
little-endian	283	nearest	296
Multiply		zero	296
Matrix	283	little-endian	
Extract	288	ceiling	296
Immediate	295	floor	296
Floating Point	303	nearest	296
Galois	308	zero	296
multiply matrix		floating-point	
complex		double	
floating-point		big-endian	303
double		little-endian	303
big-endian	303	half	
little-endian	303	big-endian	303
half		little-endian	303
big-endian	303	single	
little-endian	303	big-endian	303
single		little-endian	303
big-endian	303	switch	
little-endian	303	big-endian	311

little-endian	311
Switch	311
translate	
bytes	
big-endian	313
little-endian	313
doublets	
big-endian	313

little-endian	313
octets	
big-endian	313
little-endian	313
quadlets	
big-endian	313
little-endian	313
Translate	313

BroadMX Architecture

Key architectural features for
communications performance

August 20, 1999



Major Operation Codes

31	24	23	0
major	8	other	24

MAJOR	0	32	64	96	128	160	192	224
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								



SuperSpring

■ Decouples Access from Execution



August 20, 1999

3



SuperThread

- Simultaneous Multithreading
- Expensive resources (\$, X, E, T) shared among threads
 - ◆ improves utilization of resources
- Cheap resources (A, B, L, S) dedicated per thread
 - ◆ keeps branch latency low
 - ◆ enables multiple front-end architectures

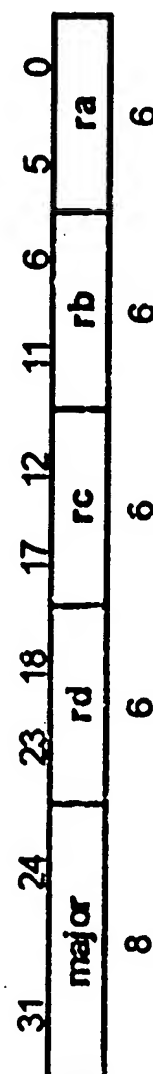
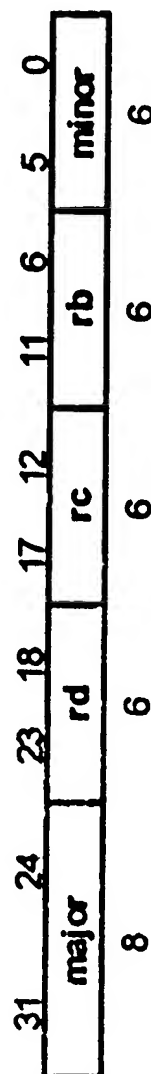
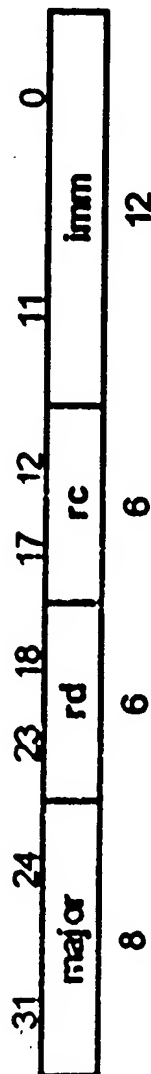
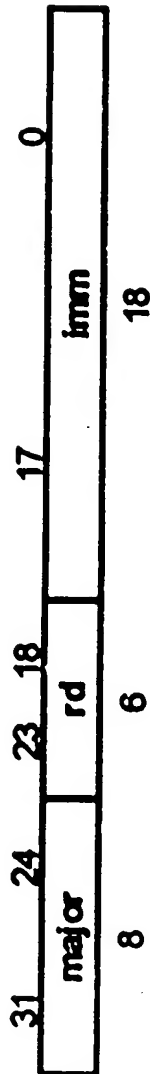
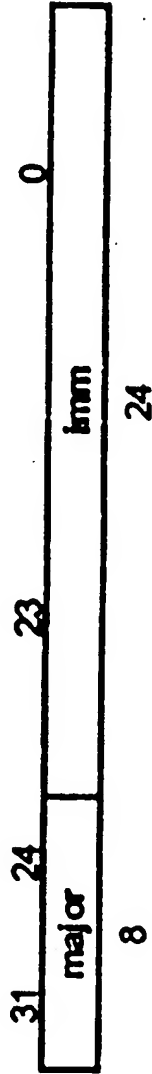


SuperWide

- Memory operand in read-only cache
- Full width register operands
- Full width register result
- Peak utilization of data path bandwidth and flexibility



Instruction Formats





Address Instructions

- Fixed-point operations over 64-bit addresses
- Add, Subtract, Set-conditional
- Boolean: 2-operand, MUX
- Shift immediate
- Shift left immediate add
- Compare



Load, Store, Sync Instructions

- **Attributes**
 - ◆ type: signed, Unsigned
 - ◆ size: 8, 16, 32, 64, 128
 - ◆ alignment: Aligned, unaligned
 - ◆ ordering: Little-endian, Big-endian
- **Synchronization: 64 A**
 - ◆ add-, compare-, mux-swap; mux
- **Addressing forms**
 - ◆ register + shifted immediate
 - ◆ register + shifted register



Synchronization

- Aligned outlet operations
 - ◆ Add-Swap
 - load mem->reg, add reg+mem->mem
 - ◆ Compare-Swap
 - load mem->reg, compare reg<->reg,
>mem if equal, store reg-
 - ◆ Mux-Swap
 - load mem->reg, mux:mask,reg,mem->mem
 - ◆ Mux
 - load mem, mux:mask,reg,mem->mem

Branch Instructions

■ B.LINK, B.LINK.I	Procedure call
■ B.I	Unconditional
■ B	Procedure return, switch
■ B.DOWN	Gateway return
■ B.BACK	Exception return
■ B.HALT	Interrupt wait
■ B.BARRIER	Instruction-fetch wait
■ Branch conditional	
■ Branch hint	
■ Branch gateway	



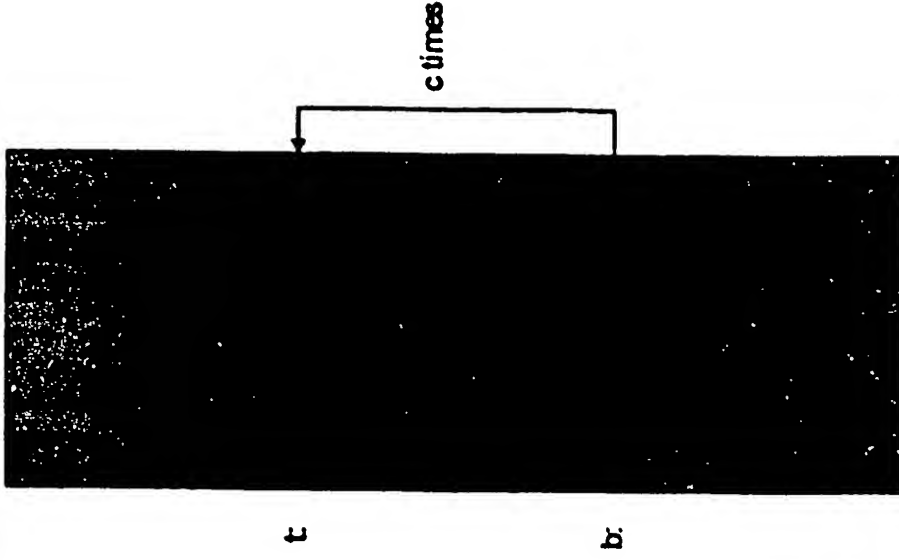
Branch Conditional

- Floating-point: F16 F32 F64 F128
 - ◆ B.E.F, B.L.G.F, B.L.F, B.GE.F
- Homogeneous Coordinates: 4xF32
 - ◆ B.V.F, B.NV.F, B.I.F, B.NI.F
 - ◆ Visible: line within viewing cube
 - ◆ Invisible: line outside viewing cube
- Fixed-point: 128 bits
 - ◆ B.E, B.NE, B.L, B.GE, B.L.U, B.GE.U
 - ◆ B.AND.E.Z, B.AND.NE.Z
 - ◆ B.E.Z, B.NE.Z, B.L.Z, B.G.Z, B.LE.Z, B.GE.Z



Branch Hint

- Hints for loops, switches, methods
- Fully interruptible
- B.HINT.l b,c,t
- B.HINT b,c,rd
 - ◆ Branch at **b** is likely **c** times, to **t/rd**, then is not likely.





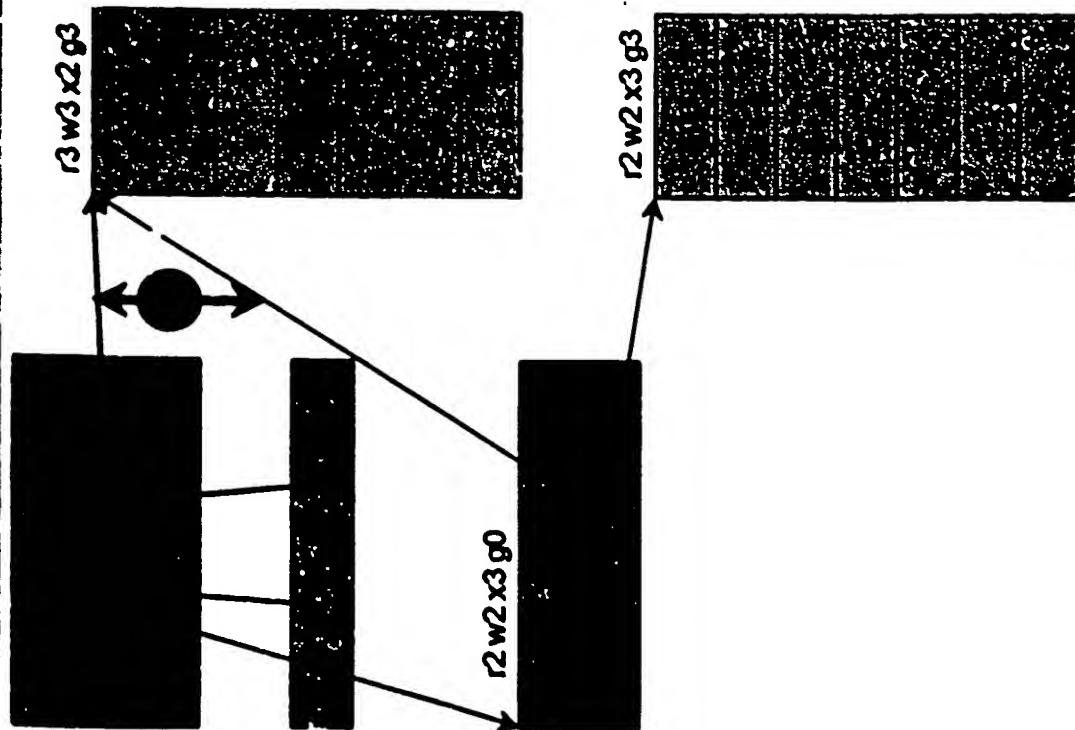
Branch Gateway

■ Gateway

- ◆ level 0 to 2
- ◆ secure entry
- ◆ data pointer
- ◆ stack pointer

■ Code

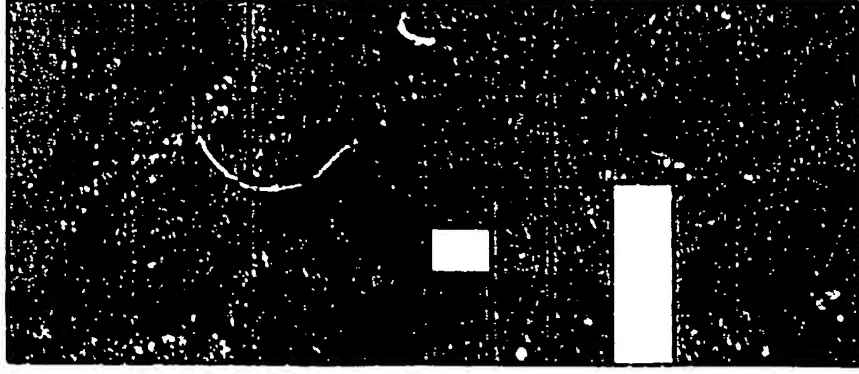
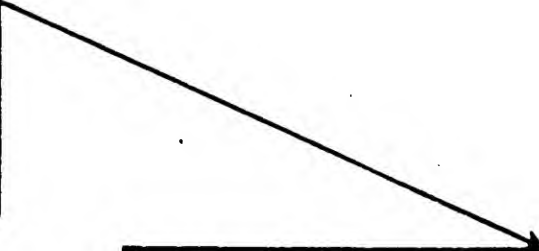
- ◆ LI64LA dp=dp,off
- ◆ LI64LA lp=dp,0
- ◆ B.GATE (lp=dp,lp)
- ◆ LI64LA dp=dp,8
- ◆ SI64LA sp,dp,off
- ◆ LI64LA sp=dp,off





Data pointer

- Memory pool for literals, statics
- procedures may share pool
- items sorted by size
- smallest items near dp
- All items aligned to size



Procedure call conventions

- Compatible with dynamic linking
- Register 63 (sp) is stack pointer
- Stack space allocated for parameters by caller
- Up to 8 parameters passed in registers 2-9
- Register 0 (lp) loaded with procedure address
- Register 1 (dp) loaded with data pointer
- To enter: BLINK lp=lp
- Register 2 contains return value
- To return: B lp



Procedure Call Structure

■ Caller (non-leaf):

ADDI	sp, -size	# allocate stack space
SI64LA	lp, sp, off	# save link pointer
SI64LA	dp, sp, off	# save data pointer
...		# use data pointer
B.LINK.I	callee	# call procedure with shared dp
...		# use data pointer
LI64LA	lp=dp, off	# load callee code address
LI64LA	dp=dp, off	# load callee data pointer
B.LINK	lp	# call procedure
...		# data pointer not available
LI64LA	dp=sp, off	# reload data pointer
...		# use data pointer
LI64LA	lp, sp, off	# reload link pointer
ADDI	sp, size	# deallocate stack space
B	lp	# return to caller

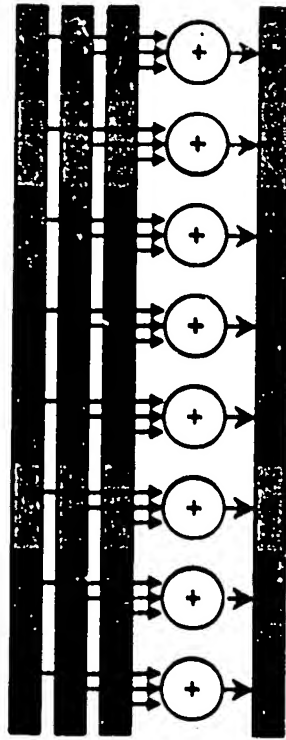
■ Callee (leaf):

...		# args in reg, use data pointer
B	0	# return to caller



Group Instructions

- Fixed-point operations over 128-bit operands with 8..128 bit symbols
- Add, Subtract, Set-conditional
- 3-operand Add/Subtract
- Add/Subtract Halve, Limiting
- Boolean: 3-operand, MUX
- Shift left immediate add
- Compare





Group triple operand

- Reduces latency for common arithmetic operations
- Group triple add/subtract
 - ◆ $rd_{128} = rd_{128} \pm rc_{128} + rb_{128}$
 - ◆ 8-128 bit symbols
- Group shift 1-4 and add/subtract
 - ◆ matches load/store with shifted index
- Group triple boolean immediate
 - ◆ $rd_i = f(rd_i, rc_i, rb_i), i=0..127$
 - ◆ 8 immediate bits specify f



Typical boolean functions

■ dcb 10000000 128

■ dc|b 11101010 234

■ d|c|b 11111110 254

■ d?c:b 11001010 202

■ d^c^b 10010110 150

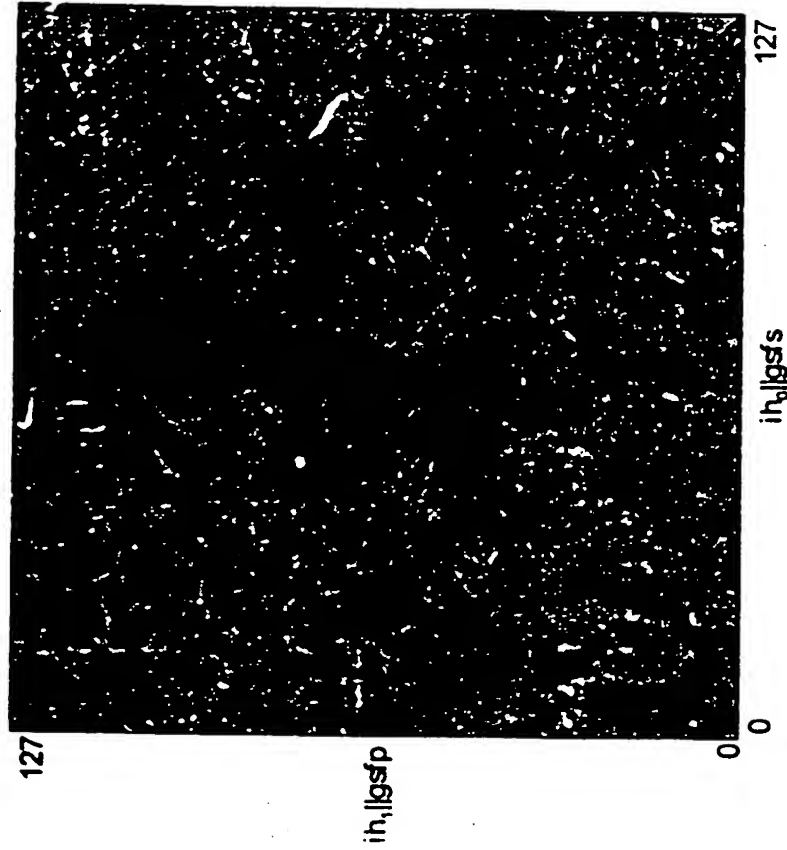
X: Crossbar Instructions

- Deposit, Withdraw
- Extract, Expand, Compress
- Swizzle, Select, Shuffle
- Shift
- Shift-Merge
- Rotate

- Wide Switch

Crossbar field

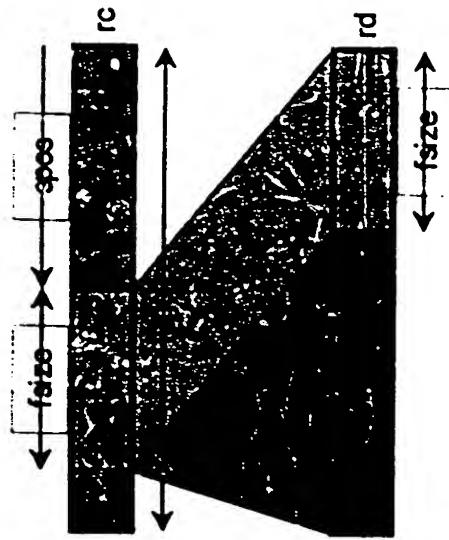
■ fsize, shift (or spos/dpos)



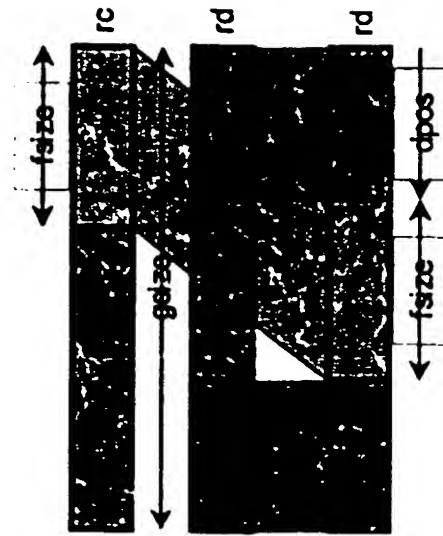
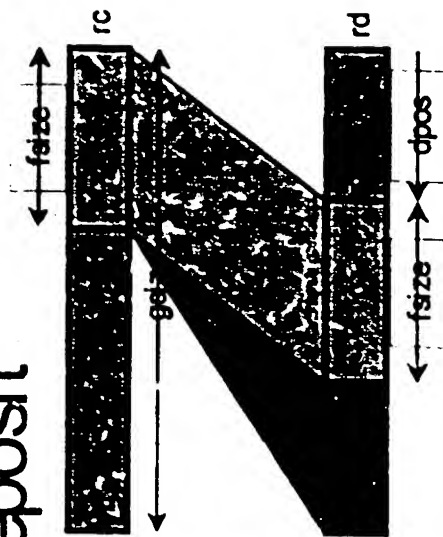


Crossbar field

■ withdraw

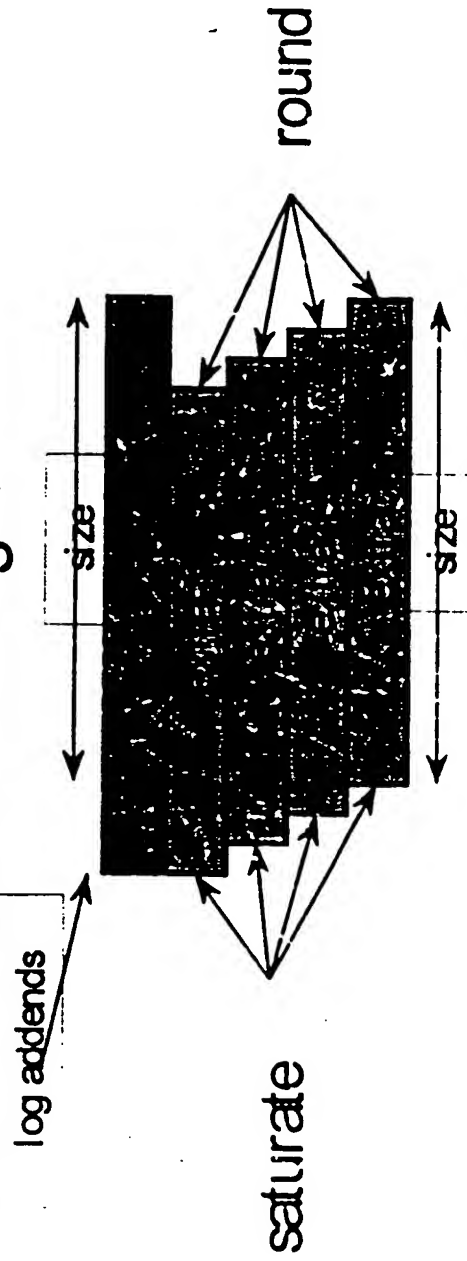


■ deposit



Crossbar extract control

- immediate control fields
 - ◆ 2 size 8, 16, 32, or 64 bits
 - ◆ 1 saturate signed, unsigned
 - ◆ 2 round floor, ceil, zero, even
 - ◆ 2 shift 0-3 bits from right





Crossbar extract

- $rd_i = (ra_{128} \parallel rb_{128})_{f(rc_{32,i})}$, $i=0..127$
- extract w/register operand control
- register specifies:
 - 8 fsize field size
 - 8 dpos destination position
 - 9 gssp group size and source position
 - 1 s signed vs unsigned
 - 1 n (real vs complex)
 - 1 m extract vs merge (or mixed sign)
 - 1 l saturation vs truncation
 - 2 rnd rounding